Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Kommunikationsnetze

Towards Data-driven Dependability Assurance for Softwarized Industrial Networks

Petra Vizarreta Paz, M.Sc.

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzende:		Prof. Dr.-Ing. Antonia Wachter-Zeh
Prüfer der Dissertation:	1.	Priv.-Doz. Dr.-Ing. habil. Carmen Mas Machuca
	2.	Prof. Kishor S. Trivedi

Die Dissertation wurde am 18.06.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 09.09.2019 angenommen.

Towards Data-driven Dependability Assurance for Softwarized Industrial Networks

Petra Vizarreta Paz, M.Sc.

09.09.2019

Abstract

The recent trend of Industry 4.0 promotes the concepts of "*industrial internet and digital factory*", requiring the enhancement of legacy industrial networks, which currently rely on closed and proprietary protocol stacks to ensure industrial grade of service. Softwarized network architectures, i.e., *Software Defined Networking (SDN)* and *Network Function Virtualization (NFV)*, can aid this transition by providing a fine-grained network traffic control and high degree of programmability, with open standards and protocols. The feasibility of achieving the industrial grade of service with SDN/NFV-based networks has already been demonstrated in the test environment. However, the dependability, which is a key requirement for the commercial adoption of softwarized networks in the mission critical applications, has been widely overlooked in state-of-the-art literature. The work presented in this thesis aims to close this gap, by providing contributions in the following four areas.

First, the analysis of the technical and economical incentives for softwarization of industrial communication networks was conducted and evaluated, in a wind park case study. The baseline of the case study was SDN/NFV-based industrial network solution tested in the operational wind park within the VirtuWind project. SDN and NFV were introduced to facilitate the tighter integration of wind parks into future Smart Grids. The capital and operational expenditures have been modelled in order to quantitatively evaluate the benefits of SDN and NFV. The case study has demonstrated that significant savings can be achieved through network softwarization, making it a promising solution to facilitate its seamless integration into the Smart Grids and further reduce the cost of wind energy.

Second, the framework for dependability assessment and forecasting based on Software Reliability Growth Models (SRGM) was developed. The framework provides guidelines for network operators to decide when a controller software is mature enough to be deployed in operational environment, based on the reliability requirements of network applications. Consequently, the operators can quantify the marginal benefits of the prolonged testing phase on the software quality. The accuracy of software reliability prediction in the early phase of the software lifecycle was improved by extrapolating the behaviour of previous controller software releases. Novel software maturity metric has been proposed, that can help operators discriminate between the competing SDN controller designs. The framework was validated in the case study on the two largest open source SDN controller platforms, Open Network Operating Systems (ONOS) and OpenDaylight (ODL), whose code and bug repositories are publicly available. Such SDN controllers are realized as distributed platforms, for scalability and high-availability reasons. Hence, the third contribution consists in analysis and modelling of the defects in such distributed control plane architectures.

The proposed framework for dependability assessment for distributed SDN controller implementations was based on Stochastic Reward Nets (SRN). The framework provides a platform for characteri-

zation of failure dynamics and user-perceived service availability in distributed SDN implementations. The preliminary analysis of the nature of software defects in ONOS and ODL bug repositories showed that the bugs in distributed implementations contribute to a significant number of the recent controller outages, which challenges the efficiency of redundancy as the primary fault tolerance mechanism. The taxonomy of software defects was provided, localizing dependability bottlenecks and contributions of each defect category. The modelling abstractions of the imperfect SDN control plane and its interaction with the service plane were provided in the formalism of SRN, which capture the relationship between the system state and dependability metrics of interest.

Fourth, a particular class of defects in distributed SDN control plane implementation, namely software ageing, was analyzed. Software ageing refers to the gradual performance degradation and resource leaks, which manifest only after the long hours of the operation. The effects of software ageing are typically mitigated by *software rejuvenation*, i.e., planned restarts, cleaning the internal system state before the performance or available resources fall below critical threshold. A framework for management of ageing in softwarized networks, has been developed and validated in the case study on open source SDN controllers. The results showed that software ageing is a systematic problem that cannot be neglected, since it stems not only from bugs, but also design trade-off in distributed network operating systems.

The dependability assurance frameworks proposed in this dissertation are the bases towards the robust, data-driven, quality assurance for softwarized industrial networks.

Kurzfassung

Der jüngste Trend von Industrie 4.0 fördert neue Konzepte zu ïndustriellem Internet und digitaler Fabrikünd zielt dabei insbesondere auf die Verbesserung der Servicequalität in Industrienetzen (Industrial Grade of Service) ab. Derzeit beruhen Industrienetze noch auf geschlossenen und proprietären Protokollen. Softwarebasierte Netzarchitekturen, Software Defined Networking (SDN) und Network Function Virtualization (NFV), können diesen Prozess unterstützen, indem sie eine fein abgestimmte Netzverkehrskontrolle und ein hohes Maß an Programmierbarkeit mit offenen Standards und Protokollen bereitstellen. Die Machbarkeit der Erreichung des Industrial Grade of Service mit SDN/NFV-basierten Netzen wurde bereits in einer Testumgebung demonstriert. Der Zuverlässigkeit, die eine wichtige Voraussetzung für die kommerzielle Einführung von softwarebasierten Netzen in unternehmenskritischen Anwendungen ist, wurde jedoch bisher zu wenig Aufmerksamkeit geschenkt. Diese Arbeit soll hierzu Beiträge liefern.

Zuerst wurden in einer Windpark-Fallstudie die technischen und wirtschaftlichen Anreize für softwarebasierte industrielle Kommunikationsnetze analysiert und ausgewertet. Der Fallstudie lag ein auf SDN/NFV basierendes industrielles Kommunikationsnetz zugrunde, das im Rahmen des von der EU geförderten Projektes „VirtuWind“ in einem Windpark getestet wurde. SDN und NFV wurden eingeführt, um die engere Integration von Windparks in zukünftige Smart Grids zu ermöglichen. Die Kapital- und Betriebsausgaben wurden modelliert, um die Vorteile von SDN und NFV quantitativ zu bewerten. Die Fallstudie hat gezeigt, dass durch softwarebasierte Kommunikationsnetze erhebliche Einsparungen erzielt werden können. Dies ist ein vielversprechender Ansatz, um eine nahtlose Integration in Smart Grids zu erleichtern und damit die Kosten für Windenergie weiter zu senken.

Weiterhin wurde das Framework für die Bewertung und Prognose der Zuverlässigkeit auf der Grundlage von Software Reliability Growth Models (SRGM) entwickelt. Das Framework enthält Richtlinien, anhand derer Netzebetreiber entscheiden können, wann eine Controller-Software ausgereift genug ist, um in einer Betriebsumgebung eingesetzt zu werden. Zuverlässigkeitsanforderungen von Netzeanwendungen bilden hierzu die Entscheidungsbasis. Damit können die Betreiber den Mehrwert längerer Testphasen für die Softwarequalität quantifizieren. Die Genauigkeit der Vorhersage der Softwarezuverlässigkeit in der frühen Phase des Software-Lebenszyklus‘ wurde durch Extrapolation des Verhaltens früherer Controller-Software-Releases verbessert. Es wurde eine neuartige Software-Reifegrad-Metrik vorgeschlagen, mit deren Hilfe Betreiber zwischen den konkurrierenden SDN-Controller-Designs unterscheiden können. Das Framework wurde in der Fallstudie anhand der beiden größten Open-Source-SDN-Controller-Plattformen, Open Network Operating System (ONOS) und OpenDaylight (ODL), validiert, deren Code- und Bug-Repositories öffentlich verfügbar sind. SDN-Controller werden aus Gründen der Skalierbarkeit und Hochverfügbarkeit als verteilte Plattfor-

men realisiert. Daher befasst sich ein weiterer Beitrag mit der Analyse und der Modellierung der Fehlerquellen in verteilten Steuerebenenarchitekturen.

Das vorgeschlagene Framework für die Zuverlässigkeitsbewertung für verteilte SDN-Controller-Implementierungen basiert auf Stochastic Reward Nets (SRN). Es bietet eine Plattform zur Charakterisierung der Fehlerdynamik und der vom Benutzer wahrgenommenen Dienstverfügbarkeit in verteilten SDN-Implementierungen. Eine vorläufige Analyse der Art von Softwarefehlern in ONOS- und ODL-Bug-Repositories ergab, dass Fehler in verteilten Implementierungen zu einer erheblichen Anzahl von Controller-Ausfällen in den letzten Jahren beigetragen haben. Damit wird die Effizienz der Redundanz als primärer Fehlertoleranzmechanismus in Frage gestellt. Eine Taxonomie von Softwarefehlern wurde erstellt, wodurch Zuverlässigkeitsengpässe und Anteile jeder Fehlerkategorie lokalisiert werden konnten. Die Modellierungsabstraktionen der unvollständigen SDN-Steuerebene und ihre Interaktion mit der Serviceebene wurden im Formalismus von SRN bereitgestellt, der die Beziehung zwischen dem Systemstatus und den interessierenden Zuverlässigkeitsmetriken erfasst.

Abschließend wurde das Altern von Software als eine bestimmte Klasse von Fehlern bei der Implementierung einer verteilten SDN-Steuerebene analysiert. Software-Alterung bezieht sich auf den allmählichen Leistungsabfall und Ressourcenlecks, die sich erst nach vielen Betriebsstunden bemerkbar machen. Die Auswirkungen der Softwarealterung werden in der Regel durch Softwareverjüngung, d.h. durch geplante Neustarts, verringert. Dabei wird der interne Systemstatus bereinigt, bevor die Leistung oder die verfügbaren Ressourcen unter den kritischen Schwellenwert fallen. In der Fallstudie zu Open-Source-SDN-Controllern wurde ein als ARES bezeichnetes Framework für das Alterungsmanagement in softwarebasierten Netzen entwickelt und validiert. Die Ergebnisse zeigten, dass das Altern von Software ein systematisches Problem ist, das nicht vernachlässigt werden kann, da es nicht nur auf Fehlern beruht, sondern auch durch einen Kompromiss im Design bei verteilten Netzbetriebssystemen verursacht sein kann.

Die in dieser Dissertation vorgeschlagenen Rahmenbedingungen für die Zuverlässigkeitssicherung bilden die Grundlage für eine solide Qualitätssicherung für softwarebasierte industrielle Netze

Contents

Acronyms

AD-SAL API-Driven SAL 15

ANN Artificial Neural Networks 64, 119

CHO Continuous Hours of Operation 103, 104, 106

CPS Cyber Physical Systems 1

CTMC Continuous Time Markov Chain 21, 71, 82

DC Docker container 85, 90

DoE Design of Experiments 98, 108

FCAPS fault, configuration, accounting, performance, security 14

FT Fault Tree 21

FW firewall 3

GoF Goodness of Fit 48, 52

HSZ Heap Size 111, 112, 113

HUS Heap Usage 111, 112, 113

IDS Intrusion Detection System 3

IED Intelligent Electronic Device 29, 30, 31

IETF Internet Engineering Task Force 118, 119

IoT Internet of Things 1, 40

ITS Intelligent Transportation Systems 1

KPI Key Performance Indicators 42, 68, 97, 109, 111, 112, 113

LOC Lines of Code 18, 19

LSE Least Square Estimation 45, 48, 100

M2M Machine-To-Machine 114

MANO Management and Orchestration 14, 19, 35

MD-SAL MD-Driven SAL 15, 16

MLE Maximum Likelihood Estimation 45, 100

MoM Method of Moments 45

MPTCP Multi Path TCP 93

MSE Mean Square Error 48, 51, 54

NBI North Bound Interface 13

NFV Network Function Virtualization 1, 2, 3, 5, 6, 7, 8, 11, 13, 14, 15, 19, 20, 21, 28, 31, 33, 34, 35, 36, 39, 40, 42, 118

NFVI NFV Infrastructure 13, 14

NH-CTMC Non-Homogeneous Continuous Time Markov Chain 44

NHPP Non-Homogeneous Poisson Process 44, 45, 46, 47, 52, 53

NLP Natural Language Processing 64, 65, 101, 119

NTP Network Time Protocol 73

ODL OpenDaylight 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 33, 34, 35, 41, 43, 44, 48, 49, 50, 51, 52, 55, 57, 61, 63, 64, 67, 68, 69, 70, 71, 72, 75, 76, 78, 80, 81, 92, 95, 96, 97, 98, 101, 102, 103, 105, 106, 107, 108, 109, 110, 112, 113, 114, 115, 117, 143, 144, 147

ONF Open Networking Foundation 33, 70, 118, 119

ONOS Open Networking Operating System 13, 15, 17, 18, 19, 20, 22, 23, 41, 43, 44, 48, 49, 50, 51, 52, 54, 55, 56, 57, 58, 61, 62, 63, 64, 67, 68, 69, 70, 72, 75, 76, 78, 80, 81, 92, 95, 96, 97, 98, 102, 103, 105, 106, 107, 108, 109, 110, 112, 113, 114, 115, 117, 143, 144, 147

OOM Out Of Memory 112

OPNFV Open Platform for NFV 14

OSM Open Source MANO 14, 35

PCA Piecewise Constant Approximation 47

PNF Physical Network Function 14

QoS Quality of Service 1, 5, 6, 28, 30, 33, 34, 40

RAM Random Access Memory 111

RBD Reliability Block Diagram 21

RNN Recurrent Neural Networks 64

RPC Remote Procedure Call 75

RSS Resident Set Size 111, 113

RTU Remote Terminal Unit 29, 30, 31

SAL Service Abstraction Layers 15

SBI South Bound Interface 13, 18

SCADA Supervisory Control and Data Acquisition 29, 30, 31, 33, 36, 37

SDN Software-Defined Networking 1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 18, 20, 21, 23, 28, 31, 33, 34, 35, 36, 39, 40, 41, 42, 43, 58, 61, 67, 68, 69, 71, 92, 95, 117, 118, 119

SFC Service Function Chaining 14, 16, 21, 22, 34

SLA Service Level Agreement 40

SoA state-of-the-art literature 4, 5, 6, 7, 8, 19, 20, 21, 28, 96, 97, 101, 117

SRE Software Reliability Engineering 119

SRGM Software Reliability Growth Models 6, 9, 23, 24, 42, 43, 44, 45, 47, 49, 51, 54, 55, 57, 58, 61, 62, 63, 64, 118, 119

SRN Stochastic Reward Nets 7, 9, 24, 68, 69, 71, 82, 84, 91, 118

SSA Steady State Availability 86, 87

TS Theil's statistics 48

TTE Time to (resource) Exhaustion 98, 99, 114, 115

TTF Time to Fail 48, 49, 51

TTR Time to Repair 48, 49, 51

VIM Virtual Infrastructure Management 14

VM virtual machine 85, 90

VNF Virtual Network Function 3, 6, 13, 14, 21, 35

VNFM VNF Manager 14

VSZ Virtual Memory Size 111

VTN Virtual Tenant Network 16, 34

WAN Wide Area Network 5, 40, 67

WSN Wireless Sensor Networks 40, 92, 119

WTG Wind Turbine Generator 29, 30, 31, 36, 37

Chapter 1

Introduction

Industrial networks have undergone significant changes in the past few decades. Started as closed systems, whose network protocols were developed independently and tailored to suit individual use cases, industrial networks have been evolving towards more interconnected systems. The need for exchange of information, as well as efficient coordination of the diverse systems is growing[1], as new integrated industrial systems have emerged, e.g., Smart Grids, or Intelligent Transportation Systems (ITS). The recent trends of Industry 4.0[2], including Cyber Physical Systems (CPS) and Internet of Things (IoT), require high degree of automation of industrial systems, their tighter coupling and an efficient coordination; more specifically:

Industry 4.0: current trend of automation and *data exchange* in manufacturing technologies, including CPS, IoT, cloud computing and cognitive computing

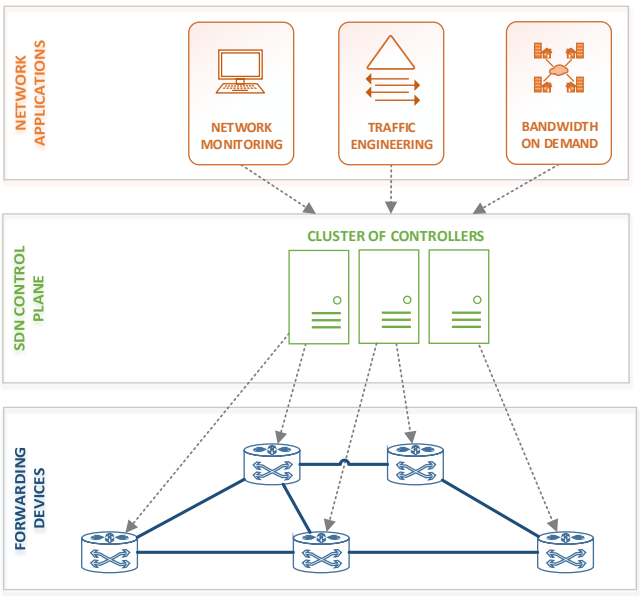
CPS: mechanism controlled or monitored by computer-based algorithms, tightly *integrated with the internet* and its users

IoT: *network of physical devices* embedded with electronics, software, sensors, actuators, and *network connectivity* which enable these objects to *collect and exchange data*

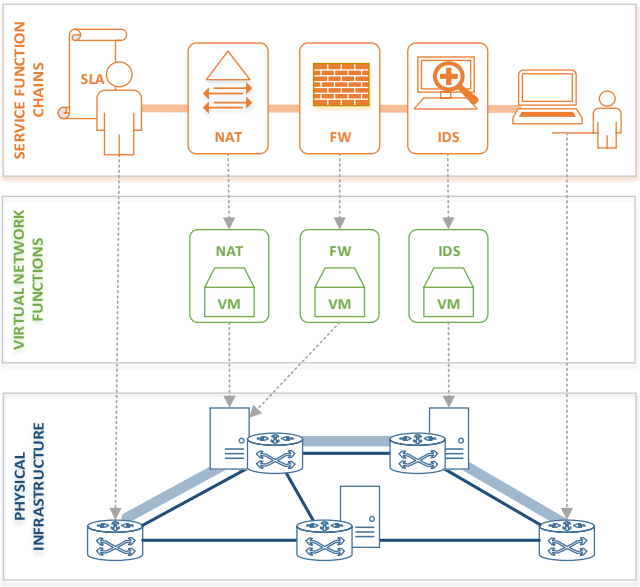
Industrial communication networks rely on the proprietary protocol stacks, and are nowadays not prepared for a seamless integration, due to the the lack of mechanisms for automated and secure exchange of information. Existing industrial networks have high configuration and management complexity, due to the diversity of network protocols and devices. Service provisioning in today's industrial networks is still a rather slow process and has to be performed by highly specialized network administrators. Upgrades and updates of the network are error prone and time consuming as they require many hours of testing. Mission critical systems, such as power plants, need to be taken out of service during the maintenance operations, which leads to a loss of revenue.

The recent concepts of network softwarization, Software-Defined Networking (SDN) and Network Function Virtualization (NFV), enable a fine grained per-flow Quality of Service (QoS) control and high degree of programmability with open and extendible protocol stack, as illustrated in Fig. 1.1.

[1] Source: German Federal Ministry for Economic Affairs and Energy (BMWI): Industrie 4.0
[2] Source: Forbes "Why Everyone Must Get Ready For The 4th Industrial Revolution?"



(a) With SDN, the distributed control plane logic of forwarding devices, i.e., switches and routers, is moved to a software entity called SDN controller, effectively decoupling the control plane (e.g., path computation) from data plane functions (i.e., switching).



(b) In NFV, higher layer network functions, such as firewalls or intrusion detection systems, which are traditionally implemented in a specialized hardware, are replaced with modular software components running on commodity hardware. Service is composed by steering the traffic through these modular software functions.

Figure 1.1: Softwarized network architectures.

SDN: With SDN, the distributed control plane logic of forwarding devices, i.e., switches and routers, is moved to a software entity called SDN controller, effectively decoupling the control plane (e.g., path computation and traffic engineering) from data plane functions (i.e., switching), as illustrated in Fig. 1.1a. The SDN controller acts as a broker between the network applications and the physical network infrastructure, providing an integrated interface towards diverse set of forwarding devices. This approach significantly simplifies the network management and augments the network programmability with standardized and open interfaces.

NFV: In NFV, higher layer network functions, such as firewall (FW) or Intrusion Detection System (IDS), which are traditionally implemented in a specialized hardware, are replaced with modular software components running on commodity hardware, as illustrated in Fig. 1.1b. These modular software components are sharing the physical resources using standard virtualization frameworks, are hence called Virtual Network Functions (VNFs). Such modular network functions can be further chained to provide composite services, offering much greater flexibility and lower cost of the service deployment for the network operators. Service orchestration, lifecycle management of VNFs and control of the physical network infrastructure are provided by open and standardized network interfaces[3].

First field trials have shown the feasibility of SDN/NFV-based networks in operational industrial environment [110], empirically proving the anticipated benefits in terms of lower cost and network management automation, through a logically centralized control. The next challenge that industrial network operators need to address is to guarantee the same or better level of performance in softwarized networks, as in highly-optimized special-purpose legacy industrial networks. The contemporary performance evaluations typically focus on the throughput and response times, while the dependability, which is the key requirement for the wide spread adoption in industrial domains is overlooked or oversimplified.

The dependability is an umbrella term for the trustworthiness of the computing system. Dependability of the system is defined in three broad aspects, *attributes*, *threats* and *means*, as illustrated in Fig. 1.2.

The formal definition of the dependability terms, used throughout this thesis, is adapted from IFIP Working Group 10.4 Dependable Computing and Fault Tolerance[4]:

Attributes: describe *the metrics to quantify system dependability*, such as availability[5], reliability[6], and maintenability[7]; Note that sometimes security attributes, such as confidentiality and integrity, are also included in dependability attributes. Since safety and security are not addressed in the scope of the thesis, their definition is omitted.

Threats: describe *the factors that affect system dependability*. Although the terms fault, error and failure are often used interchangeably in everyday speech, they have different meaning in the

[3] ETSI Network Functions Virtualisation (NFV) https://www.etsi.org/technologies-clusters/technologies/nfv

[4] IFIP Working Group 10.4 Dependable Computing and Fault Tolerance https://www.dependability.org/wg10.4/

[5] Availability: the probability that a *repairable system or system element is operational* at a given point in time under a given set of environmental conditions.

[6] Reliability: defined as the probability of a system or system element performing its intended function under stated conditions *without failure for a given period of time*

[7] Maintenability: defined as the probability that a system or system element *can be repaired in a defined environment within a specified period of time*

context of dependable systems. *Fault* is a system defect, e.g., a software bug, the initial root cause of the failure. *Error* is an abnormal behaviour, e.g., a system state that activates a bug. Without appropriate and timely activation of fault tolerance mechanisms, an error, i.e., an incorrect system behaviour, will be perceived by a user as a *failure*.

Means: describe *the ways to improve system dependability*, such as fault prevention, fault removal, fault tolerance and fault forecasting.

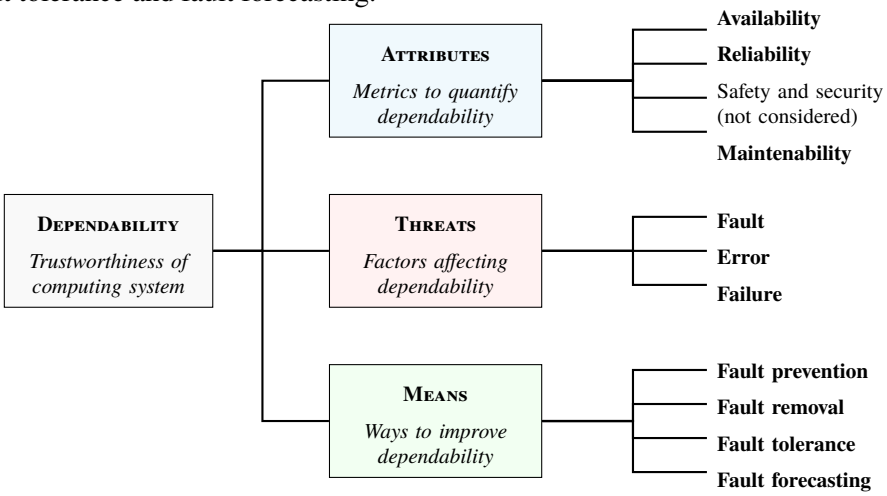


Figure 1.2: Three dimensions of dependability (adapted from IFIP Working Group 10.4).

The key limitations of the related work on dependability of softwarized networks can be summarized along these three dimensions:

Attributes: Generic dependability attributes, such as operational probability, are not sufficient to precisely describe software behaviour. The effects of long term *reliability growth* due to the software maturity and short-term *reliability degradation* due to resource leaks are not precisely captured in generic reliability metric.

Threats: The existing work has focused on network hardware failures, such as random link and switch failures, while *software failures have been neglected or oversimplified* so far. Given that many of the major network platforms, ranging from packet I/O to management and orchestration, are open sourced[8], a detailed analysis of dependability threats can be carried out by mining valuable data provided by public software repositories.

Means: The measures to improve dependability of softwarized networks in the state-of-the-art literature (SoA) have focused mainly on fault tolerance and structural protection, i.e., simple redundancy. While simple component replication may be an efficient in case of independent hardware failures, it is not as efficient in the case of software failures. This happens due to shared software defects, state synchronization overhead between the replicas, as well as faulty failure contention procedures, which might introduce new failure modes. Moreover, *fault forecasting, prevention and removal* have been widely overlooked in the context of softwarized networks.

The limitations of the related work on dependability of softwarized networks are further discussed in Sec. 2.3.1.

[8]The Linux Foundation: Open Source Networking https://www.linuxfoundation.org/projects/networking/

1.1 Research Challenges

Network softwarization is the necessary step in the evolution towards the next generation industrial networks, and dependability is the key feature for the industrial applications. Hence, it is of the utmost importance to develop the frameworks to accurately estimate the dependability of all of layers in softwarized networks. The main goal of this thesis is to advance the SoA understanding of dependability of softwarized networks for industrial applications.

RQ1: Feasibility analysis of softwarized industrial networks

The first objective of the thesis is to assess the techno-economic feasibility of softwarized industrial networks, which has not been addressed so far in the SoA. While the benefits of SDN/NFV-based networks, such network programmability and fine-grained QoS control are widely addressed in the context of data centers and service providers, very few studies have addressed the actual incentives for softwarization of industrial networks. The techno-economic analysis aims to provide a qualitative and quantitative feasibility study on: i) *technological incentives*: assessing whether an industrial grade of performance be achieved with SDN/NFV-based network solutions, and ii) *economic incentives*: providing cost models to translate the benefits of SDN/NFV-based networks to tangible savings for industrial network operators.

RQ2: Characterization of failure dynamics in softwarized networks

The reliability of the hardware follows a well-known bathtub curve. However, the software failure dynamics, which has an entirely different pattern, is far less studied. In the long term, the reliability of the software (*release*) grows with time, due to the removal of defects and *software maturity*. In the short term, the reliability of the software (*instance*) degrades, due to the resource leaks, as well as the natural increase in the memory consumption, which is an effect known as *software ageing*. Providing high-fidelity stochastic models for the interplay between these two factors is crucial for an accurate failure forecasting.

RQ3: The efficiency of fault tolerance in softwarized networks

A simple replication is not always efficient in case of software failures, as it can only provide the environmental diversity counteracting some of the transient failures, while deterministic failures, such as an error in the path computation module, are shared between the replicas [48]. Moreover, many of the network functions are stateful, introducing an additional overhead of synchronization of the replicas. For an example, in SDN, network programmability is enabled through a *logically centralized* control plane. Production networks deploy multiple *physically distributed* SDN controllers for scalability and reliability reasons, which in turn rely on distributed consensus protocols to operate in logically centralized manner. Bugs in a distributed control plane system can have disastrous effects on the data plane traffic, such as loosing the traffic by installing paths containing blackholes or loops. Practical experience reports on large-scale SDN deployments[9], show that high-availability issues prevail, which is an effect that has been widely neglected in the SoA.

[9]Based on the practical experience report [63] on B4 [86], Google's internal Wide Area Network (WAN), carrying the traffic between data center clusters, which is arguably the biggest live SDN network. Report showed that *control plane software failures* prevail, *maintaining globally consistent network state* is a difficult, and the *cascade of control-plane element failures* is a common culprit of critical customer impacting failures.

RQ4: User-perceived service availability in softwarized networks

In softwarized network architectures, such as SDN and NFV, the entire control plane intelligence is concentrated in network orchestration platforms. However, control plane services are not needed continuously, but just while the service requests are being processed. Depending on the service, the control plane availability will be sampled at different times, i.e., at *request arrival time*, and for a different duration, i.e., during *request serving time*. The relationship between control plane failure dynamics, i.e., downtime distribution times, and service characteristics will have a crucial impact on the user-perceived service availability, which is not described precisely by general availability and reliability metrics.

1.2 Main Contributions

The key contributions of this dissertation are summarized in this section. The author's relevant publications have been indicated in the brackets, as well as the mapping to the research questions.

C1: Techno-economic analysis of softwarized industrial networks [8, 18, 19, 7] (**RQ1**)

The analysis of technological and economic incentives for softwarization of industrial networks has been analysed in the case study of a wind park. An SDN/NFV-based industrial network prototype deployed in an operational wind park within the European project VirtuWind [110], has provided an insight into operational details of production industrial networks, enabling a realistic assessment of feasibility of softwarized industrial networks. The analysis has shown that the main benefits are achieved by providing the protocol openness and fine-grained QoS control in three domains: i) replacing proprietary Industrial Ethernet switches with commodity SDN-enabled forwarding devices[10] and ii) replacing the proprietary monolithic security appliances, with modular open-source VNFs [8], and iii) automated service provisioning and network management open source network orchestration platforms. A case study of a typical wind park showed that the reduction of the cost of the access switches in the wind turbine contributes most to the CAPEX savings, the highest cost reduction is OPEX due to the shorter interruptions of the power production [19, 7].

C2: Assessing the Software Maturity with Reliability Growth Models [5, 17] (**RQ2a**)

A framework to assess and forecast the maturity of software releases, based on the Software Reliability Growth Models (SRGM), has been proposed. The framework addresses the effect of reliability growth in network control software, i.e., SDN orchestration platforms, which has been neglected in the SoA. SRGMs model the stochastic behaviour of bug manifestation and correction processes, which facilitates analysis of the long term variations in controllers' reliability. The empirical data is gathered from open source bug repositories, and the best SRGM to describe its stochastic behaviour is selected and parametrized. Having an accurate stochastic model enables the evaluation and forecasting of software reliability metrics, such as residual bug content and failure intensity, facilitating the network management decisions, such as optimal software release and adoption time. The early predictive power of SRGMs is improved by leveraging the *transfer*

[10]The properties of deterministic Ethernet, i.e., hard delay guarantees, are achieved through logically centralized queue-level flow management using the solution proposed by Guck and Van Bemten [65]

learning, i.e., learning from the behaviour of similar controller software releases. Furthermore, a novel software maturity metric is proposed, serving as a fair comparison criteria between competing software releases, when the reliability is the main concern.

C3: Dependability Assessment Framework for Distributed SDN [14, 6] (**R3,4**)
A framework to assess and forecast the maturity of software releases, named DASON, based on the data-driven Stochastic Reward Nets (SRN), is proposed. The framework includes the analysis of prevalent failure modes in practical distributed SDN implementations, as well as the modelling abstractions to assess the efficiency of redundancy in the context of softwarized networks. The assumption about perfect failover between identical software replicas and fault-free implementation of distributed protocols, often made in the SoA, is challenged. The first part provides a comprehensive analysis based on open code and bug repositories of production grade distributed SDN platforms. The analysis shows the variety of failure modes that have been overlooked in the related work, e.g., *resource leaks* and *failure contention*. In the second part, the modelling abstractions for the identified failure modes are provided. Dependability models, in the formalism of SRN, are used to characterize the control plane failure dynamics, as well as the impact on the *user-perceived service availability*. Furthermore, an application of data-driven SRN for the network management is demonstrated, e.g., as a tool for the operators and network architects to compare different deployment scenarios and optimize preventive maintenance policies.

C4: Management of Software Ageing and Rejuvenation in SDN [4] (**RQ2b**)
A framework for management of software ageing and rejuvenation in SDN, named ARES, is proposed. The framework addresses the problem of a short term reliability degradation due to the effects of software ageing, i.e., gradual performance loss and cumulative effects of resource leaks, which has been overlooked so far in the SoA on performance and dependability assessment of SDN platforms. The ageing defects and their common manifestation patterns have been identified based on the open bug repositories, and empirically proven in a measurement based study. Modelling of a workload-ageing relationship enables network architects and operators to predict which applications, i.e., service mix and load levels, will be affected by the effects of software ageing and up to which degree. Preventive software rejuvenation policies for mitigation of the effects of software ageing in an operational environment have been designed and discussed.

Other author's publications are only briefly mentioned in the thesis (Chapter 2). The first studies on interplay between software and network dependability in softwarized networks have been presented in [2, 15] and in [13]. Different design strategies for performant SDN-based satellite network have been proposed and benchmarked in [3, 10], while QoS-aware resource management and service composition algorithms in NFV have been addressed in [11]. A magnitude and importance of software failures has been presented in a short survey on disaster-resilient SDN [9]. The failure dynamics in network control software is addressed in more detail in a book chapter "Resilient Communication Services Protecting End-user Applications from Disaster-based Failures (RECODIS)" [1].

1.3 Thesis Outline

The overview of the dissertation is illustrated in Fig. 1.3, outlining the structure and mapping the main contributions of the thesis to the corresponding chapters.

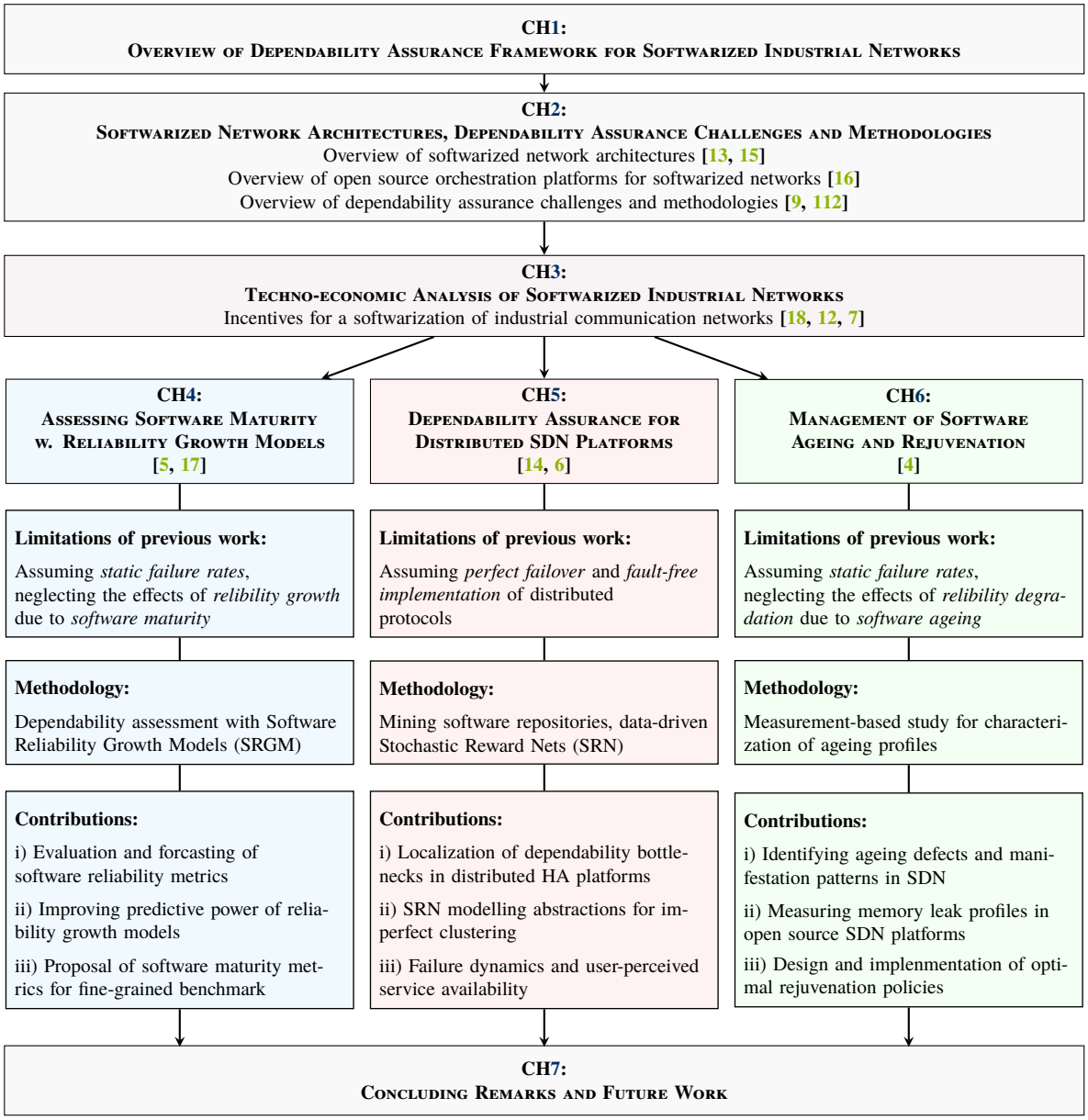


Figure 1.3: Outline of the thesis: main contributions are mapped to the corresponding chapters

Chapter 1 introduces the dissertation topic, presenting the motivation and defining the problem scope and research challenges in dependability assurance for softwarized industrial networks, followed by an overview of the key contributions.

Chapter 2 gives a background on softwarized network architectures, i.e., SDN and NFV, and provides an overview of the design and implementation of today's network orchestration platforms. The dependability assurance challenges critical for industrial communication networks are identified, followed by an overview of the SoA dependability assurance frameworks.

Chapter 3 presents a techno-economic study on softwarized industrial networks. Incentives for softwarization of industrial networks, i.e., the practical technological benefits and the magnitude of cost savings, are illustrated in a case study on the wind park communication networks.

Chapter 4 presents a framework for the assessment of software maturity with SRGM, providing a tool to model and forecast long term variations of reliability at the level of software release. The applications of the framework on the management of softwarized networks are illustrated in the case study of two largest open source SDN orchestration platforms.

Chapter 5 addresses the efficiency of redundancy in the context of softwarized networks, by studying the dependability of real-life distributed SDN control plane implementations. Dependability bottlenecks in distributed SDN architectures are identified by mining open software repositories, and modelled using SRN. The proposed models are then used to characterize the failure dynamics and evaluate user-perceived service availability.

Chapter 6 presents a measurement-based study on the effects of software ageing, i.e., short term degradation of software reliability due to the resource leaks, in SDN orchestration platforms. First, the sources of software ageing and their manifestation patterns in SDN are analyzed. The control stress tests are then designed and conducted to empirically prove that the software ageing effects have a non-negligible impact on the network performance. Finally, the preventive software rejuvenation policies are then introduced as an efficient way to mitigate the ageing effects in a production environment.

Chapter 7 concludes the dissertation with the summary and discussion of the results, providing a broader overview of the expected impact of the findings presented in this thesis, as well as the remaining open questions and outlook for future work.

Chapter 2

Background

This chapter presents an overview of softwarized network architectures (Sec. 2.1), production grade orchestration platforms for softwarized networks focusing on their dependability issues (Sec. 2.2) and dependability assurance in softwarized networks (Sec. 2.3).

2.1 Softwarized Network Architectures

The recent trend of network softwarization with SDN and NFV suggests a radical shift in the implementation traditional network intelligence, decoupling the network functionality from the hardware. This section presents an overview of architectural concepts, functional split, as well as several open source implementations of the network orchestration platforms.

2.1.1 Software Defined Networking (SDN)

With SDN, the control plane logic of forwarding devices, i.e., switches and routers, is extracted and moved to an entity called SDN controller, which acts as a broker between the network applications and physical network infrastructure. The functional split between *data*, *control* and *application* plane in SDN is illustrated in Fig. 2.1.

2.1.1.1 Data Plane

In SDN, distributed control plane logic of forwarding devices, e.g., path computation, is implemented in a logically centralized control plane, i.e., SDN controllers. The SDN forwarding devices are simple programmable devices, whose forwarding tables are populated by an SDN controller. OpenFlow has become de facto language to program the forwarding tables in SDN.

The forwarding tables consist of *rules*, *actions* and *statistics*. In OpenFlow 1.0, the rules represent a 12-tuple matching field using packet header data, such as MAC address or TCP port, as illustrated in Fig. 2.1. The matching fields can be populated with wildcards and ordered by priorities, facilitating the realization of more complex traffic steering functions than legacy IP-destination based routing. After the matching rule has been found, the actions describe the packet treatment, e.g., forwarding to a particular set of ports or packet header modification. Statistics enable simple network sensing and monitoring.

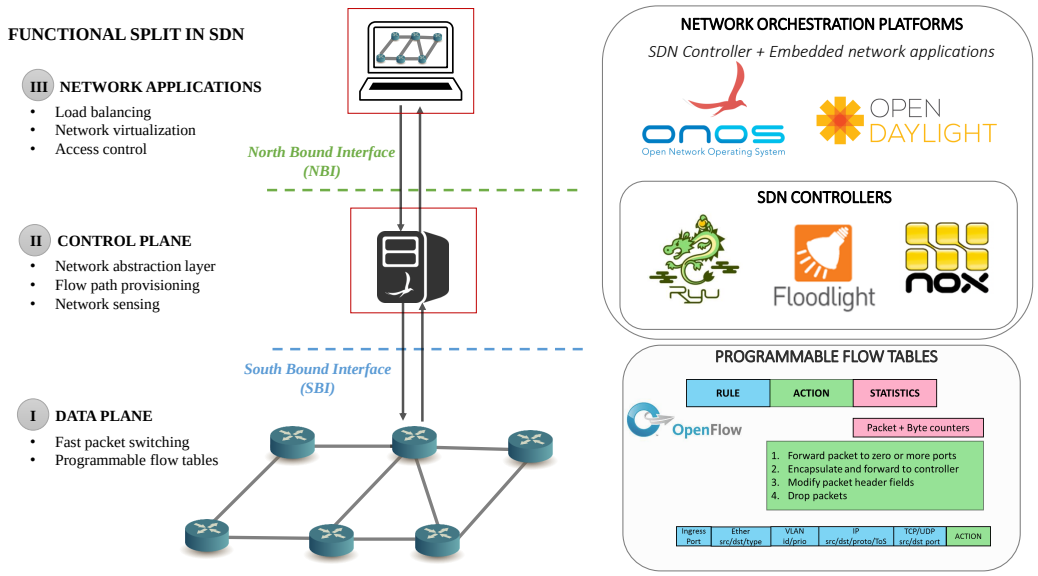


Figure 2.1: Functional split in SDN: decoupling control and data plane of L2-L4 forwarding devices.

Providing such standardized and open interfaces towards the network components, allows the network operator to avoid the vendor lock-in, and hence, to achieve lower prices of the network components thanks to the increase of the market competitiveness. SDN forwarding devices are simple programmable devices, implementing fast packet processing switching, and are cheaper than the equivalent legacy devices.

2.1.1.2 Control Plane

In SDN, basic control plane tasks, such as *network abstraction*, *flow path provisioning*, and *network sensing* are outsourced to an SDN controller.

Network abstraction. The SDN controller assumes the role of network operating system, providing an integrated interface towards a diverse set of forwarding devices, offering an abstract view of the network to the network applications, which can install policies without minding the low level implementation details.

Flow path provisioning. The SDN controller computes the path on the abstracted network topology graph. Most controller implementations support different kinds of unicast and multicast routing algorithms (e.g., Dijkstra, k-shortest paths) and policies (e.g., least cost, delay constrained). Once the abstract flow path is computed, it is compiled to a set of the flow rules and is programmed into the forwarding tables of the devices.

Network sensing. Another task of the SDN control plane is network sensing and monitoring. The statistics are collected per switch port level, as well as at the level of the individual forwarding rules. The network sensory data can be used to monitor the health of the network, triggering self-healing actions, e.g., flow re-routing upon a link failure, as well as an input for different traffic engineering policies.

Since the inception of SDN, a multitude of the controllers have emerged. The basic functionalities of an SDN controller are implemented in several open source controllers, e.g., Ryu, Nox, Floodlight. The production grade platforms, such as OpenDaylight (ODL) and Open Networking Operating System (ONOS) also provide a multitude of embedded network applications, necessary for the control, management and orchestration of the operational networks.

2.1.1.3 Application Plane

Network applications consume the data provided by SDN control plane, providing more complex services such as load balancing, management of security policies (e.g., access control), traffic engineering (e.g., bandwidth calendaring), as well as network virtualization and slicing.

Unlike OpenFlow at the South Bound Interface (SBI), there is a variety of North Bound Interface (NBI) protocols and interfaces, e.g., REST, RESTCONF, NETCONF, AMPQ.

2.1.2 Network Function Virtualization (NFV)

In NFV, higher layer network functions (e.g., firewall, DPI) are realized as software modules running on commodity hardware. These modular functions can be provisioned and chained on-demand, enabling fast instantiation of new services, as well as the resource pooling. The functional split between *packet processing functions handling user traffic* and *infrastructure orchestration and management* in NFV is illustrated in Fig. 2.2.

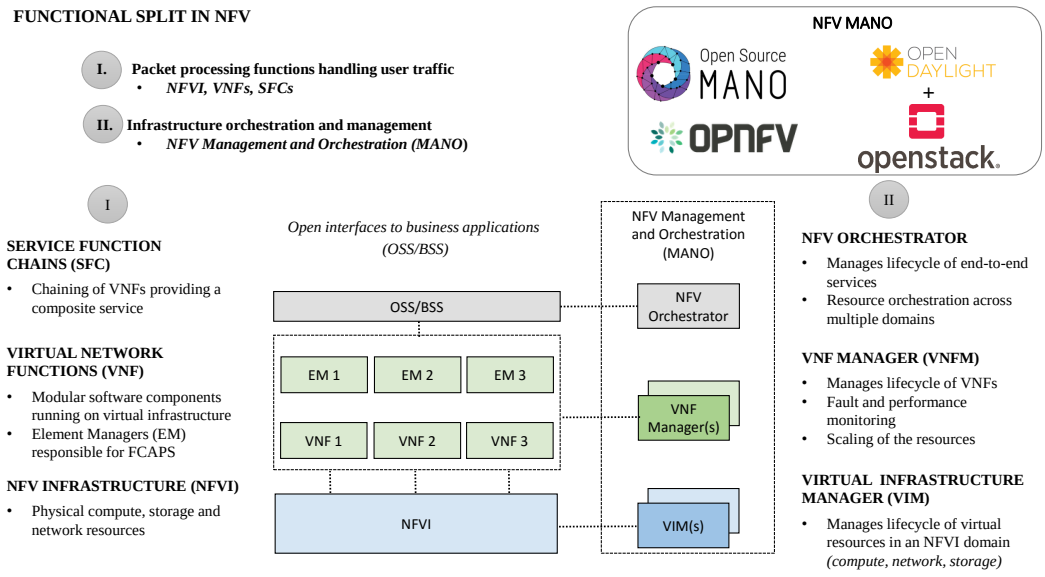


Figure 2.2: Functional split in NFV: virtualization of L4-L7 packet processing functions.

2.1.2.1 Packet Processing Network Functions Handling User Traffic

The modular *Virtual Network Functions* (VNFs), handling the user traffic, require supporting NFV Infrastructure (NFVI), i.e., physical compute, storage and networking resources, enable an efficient

resource pooling. The user traffic is steered through the ordered set of VNFs, called Service Function Chaining (SFC), offering a flexible service provisioning.

2.1.2.2 Infrastructure Orchestration and Management

NFV Management and Orchestration (MANO) is responsible for the infrastructure orchestration and management. Virtual Infrastructure Management (VIM) handles the lifecycle of virtual resources in a single NFVI domain, while VNF Manager (VNFM) manages the lifecycle of the packet processing functions, as well as *fault, configuration, accounting, performance, security (FCAPS)* management. NFV orchestrator manages the lifecycle of end-to-end services, i.e., SFC, across multiple domains.

Open Source MANO (OSM) and Open Platform for NFV (OPNFV) are the open source reference MANO implementations, while some basic functionalities can be realized with network orchestration platforms (ODL) and cloud management software (OpenStack).

2.1.3 The Role of SDN in NFV

The two described softwarized architecture concepts, SDN and NFV, are often deployed together. Different SDN/NFV-based architectures are possible, as described by ETSI NFV[1] and SDN IEEE[2].

For instance, a report on ETSI NFV architectural framework discusses several SDN controller positions: i) at VIM, ii) as managed VNF, iii) as a part of NFVI, or OSS/BSS, or v) at a separate Physical Network Function (PNF). The industrial controller prototype, deployed in operational wind park VirtuWind [110, 162], discussed in Chapter 3, is SDN-centric. The interfaces between SDN and NFV in SDN-centric architecture are illustrated in Fig. 2.3, adapted from ETSI NFV. The implementation of these interfaces in VirtuWind controller is discussed in more detail in Sec. 3.3.3.

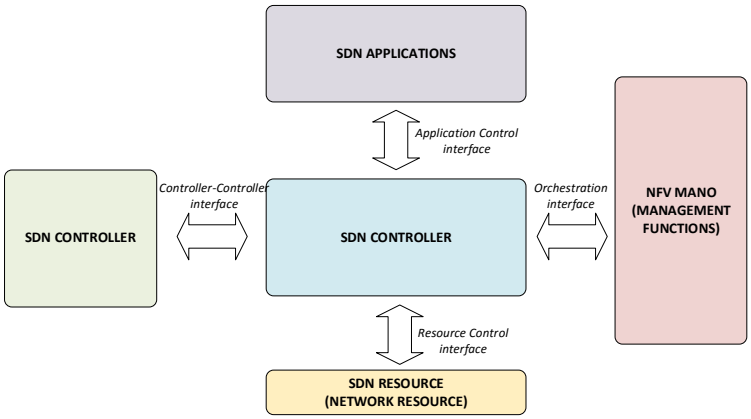


Figure 2.3: SDN-NFV interfaces proposed by ETSI NFV (adapted from report on "SDN in NFV Architectural Framework" by ETSI NFV).

[1] Report on SDN Usage in NFV Architectural Framework (ETSI NFV)
[2] SDN in NFV Architectural Framework (SDN IEEE)

2.2 Open Source Network Orchestration Platforms

Next, the overview of the two largest open source network orchestration platforms, ODL and ONOS, is presented. These two production grade network orchestration platforms implement not only the functionalities of the SDN controllers, but additionally provide support to legacy network protocols and hybrid devices, advanced security features, automated bootstrapping, as well as interworking with NFV orchestration platforms and cloud management systems. Their code internals and bug repository are publicly available, providing a rich data set for an in-depth dependability assessment. The relevance of ODL and ONOS platforms is even higher, given that they provide the code base of many commercial controllers, such as Cisco, Brocade, Huawei and Ericsson[3].

The overview of ODL architecture is adapted from authors work published in [16].

2.2.1 OpenDaylight (ODL)

The ODL controller platform is a collaborative "community-led and industry-supported framework", foreseen from the beginning to be the Linux of the networks [119]. The majority of the ODL key partners are vendors, and the initial focus was on the applications in data centers and the coexistence with network virtualization technologies. The controller size has reached 3,920,556 lines of code, with 1,210 developers from industry and research contributing to its code base, mainly written in Java. Nine releases, each one with several stability releases (SR), have been distributed between February 2013 and May 2019.

The complex code base is organized in 95 projects. Due to the space limitations, only the 55 most relevant projects covering more than 98% of the bug content are presented. In order to grasp easily the code organization, the projects are grouped into 5 categories. Descriptions of the projects are adapted from the ODL documentation[4], ranging from core controller project to advanced embedded controller applications, as illustrated in Fig. 2.4.

2.2.1.1 Core Controller Functions

This category consists in core *Controller project*, and two related projects, *topology processing (topoproc)* and *L2-switch*. As the controller project is the largest and the most important of ODL platform, its sub-components are also presented. The role of `Service Abstraction Layers (SAL)` is to decouple network application interfaces from south-bound protocol plug-ins, e.g., OpenFlow. The initial solution was `API-Driven SAL (AD-SAL)`, aiming to provide a collection of direct application interface adaptations, which evolved to a more generic `MD-Driven SAL (MD-SAL)`[5]. `MD-SAL` is providing the supporting functions for other projects. As part of the controller module, the `MD-SAL` is connecting the protocol plug-ins to the Network Function Modules[6], such as Flow Rule Manager (FRM), Topology Manager, Switch Manager, etc. Controller `clustering` enables the load sharing between a group of the controllers, as well as the fault tolerance. The `config subsystem` provides a uniform way to express configuration and requirements on other services. `NETCONF` is an XML-based

[3] Cisco Open SDN Controller, Brocade Vyatta Controller, Ericsson Cloud SDN, Huawei Agile Controller
[4] OpenDaylight project list https://goo.gl/8SfCc9
[5] OpenDaylight `MD-SAL` https://goo.gl/RfCXd9
[6] Brocade Vyatta Controller https://goo.gl/itMBX7

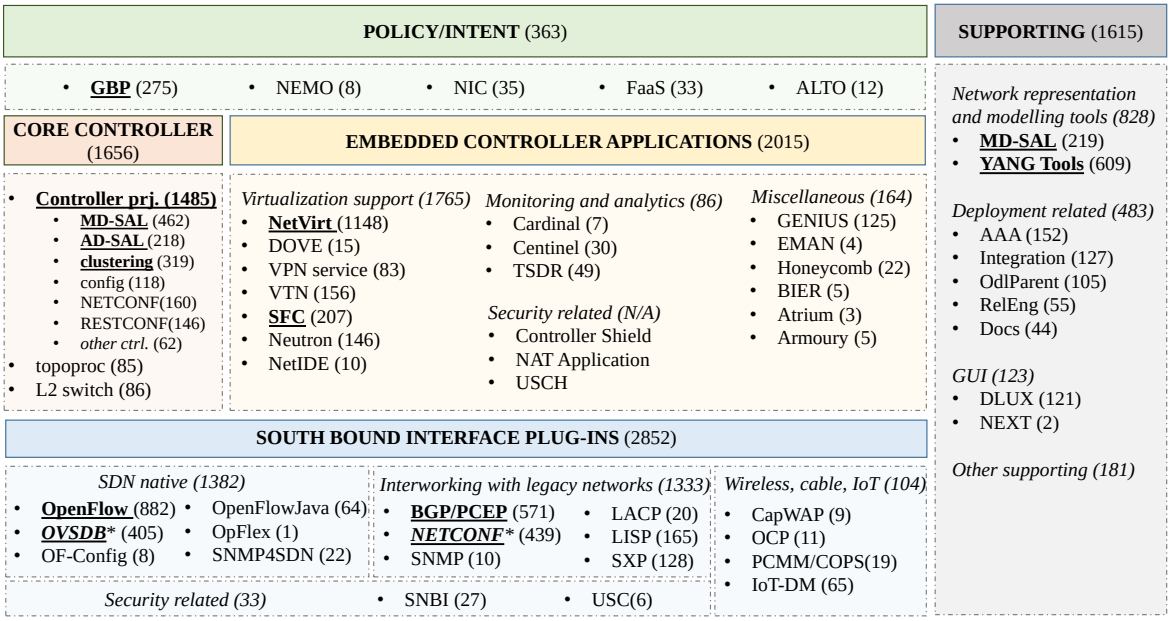


Figure 2.4: Contributions of different functional blocks and individual projects to the total bug content of the ODL platform [16] (©2019 IEEE).

protocol used for configuration and monitoring devices in the network. ODL supports the NETCONF protocol as a northbound server as well as a southbound plugin. `RESTCONF` allows access to MD-SAL data store in the controller.

2.2.1.2 Embedded Controller Applications

The ODL platform provides a multitude of embedded applications related to the original virtualization use case, as well as applications related to production environment requirements, such as monitoring, analytics and security.

Virtualization support: The *NetVirt* is a network virtualization solution that includes the support for software and hardware switches, L3VPN (BGPVPN), NAT and Floating IPs, IPv6, Security Groups, MAC and IP learning, etc. The *Distributed Overlay Virtual Ethernet (DOVE)* and *VPN service* projects have been deprecated and split into different projects, mainly NetVirt. The *Virtual Tenant Network (VTN)* is an application that provides multi-tenant virtual network on an SDN controller. The *SFC* provides ability to define and connect ("chain") an ordered set of network functions realizing a composite service; while the *Neutron* enables the integration with OpenStack Neutron networking service. The *NetIDE* provides the virtualization of SDN networks where users can bring their own controllers.

Monitoring and analytics: The *Cardinal* enables monitoring of ODL and underlying network as a service; while the *Centinel* provides a framework to collect, aggregate and sink streaming data, leveraging the *Time Series Data Repository (TSDR)*.

Security: The issues related to the security applications, such as *Controller Shield*, *NAT application* and *Unified Secure Channel (USCH)*, are not reported in the ODL bug repository.

Miscellaneous: The *Generic Network Interface, Utilities and Services (GENIUS)*, allows the interference-free co-existence with different applications, while the *Energy Management (EMAN)* implements energy measurement and control features. Other representative embedded applications are the *Honeycomb* Virtual Bridge Domain (VBD) vector packet processing, the *Bit Indexed Explicit Replication (BIER)* architecture for the forwarding of multicast data packets, the *Atrium* open source BGP Peering Router and the *Armoury* framework to request network function from workload managers.

2.2.1.3 Network Abstractions (Policy/Intent)

Network abstractions are provided to users and applications, which can specify high level policies (intents) without minding the low level hardware-specific implementation details. The *Group Based Policy (GBP)* projects allow users to express the network configuration in a declarative versus imperative way. The *Network Modelling (NEMO)* project aims to simplify the usage of network by providing a new intent northbound interface (NBI), enabling network users/applications to describe their demands for network resources, services and logical operations in an intuitive way. The *Network Intent Composition (NIC)* project enables the controller to manage and direct network services and network resources based on describing the intent for network behaviours and network policies. The *Fabric as a Service (FaaS)* project aims to create a common abstraction layer on top of a physical network, so northbound API or services can be easier to be mapped onto the physical network as concrete device configuration. The *Application Layer Traffic Optimization (ALTO)* is an IETF protocol RFC 7285, which provides simplified network views and services, e.g., cost maps, to applications.

2.2.1.4 South Bound Interface (SBI) Plugins

ODL supports a variety of southbound protocols, or plugins, adapting to the different types of networks. These plugins represent the drivers for the controller to communicate with the network devices, and represent the largest part of the code base. The SBI plug-ins are classified into: i) native to SDN OpenFlow, ii) interworking with legacy network protocols to ensure the support for hybrid networks, iii) and domain specific, such as support for wireless access points, remote radio heads, packet cable and IoT data manager, and iv) security related, such as *Secure Network Bootstrapping Infrastructure (SNBI)* and *Unified Secure Channel (USC)*.

2.2.1.5 Supporting functions

This category comprises the projects that are implicitly related to all previous categories, such as network representation and modelling tools (*MD-SAL* and *YANG tools*); deployment related functions including the standard *Authentication, Authorization and Accounting (AAA)*, release management and integration, as well as documentation; and Graphical User Interface (GUI) *DLUX* and *NEXT*. The remaining 40 projects contribute to approximately to 2% of the bug content, and are grouped together as other supporting functions.

2.2.2 Open Network Operating System (ONOS)

The focus of ONOS since its inception has been on providing scalability, high availability and carrier-grade performance fulfilling the requirements of large operator networks [28]. The project is supported

by the key telecom and data center operators, as well as network equipment vendors, such as AT&T, Google, Ericsson, Cisco, just to name the few. Overall, more than 300 developers from more than 60 organizations have contributed to its code base. The code has been written mostly in Java and contains 852,570 lines of code. New ONOS releases are distributed every quarter, which provides a steady feature development through incremental upgrades of the code base.

The architecture of ONOS is illustrated in Fig. 2.5[7]. ONOS architecture consists of functional tiers[8], which are aligned with the SDN layers:

- *Distributed core:* Since the initial scope of ONOS was developing a scalable and performant controller for service providers, the distributed core has been a part of its design since the first release. ONOS core offers a rich set of distributed primitives for representation of network state, e.g., flow statistics, optimized for their specific access patterns. A support for distributed operation in ODL started only in the later releases (see Sec. 5.3 for a comparison of the two distributed implementations);
- *Providers:* The providers implement interfaces between protocol-agnostic core and protocol-specific SBI API towards network elements. The protocol-aware providers are responsible for interaction with the environment, implementing different SBI control and configuration protocols, and collecting device specific sensory data;
- *Applications:* ONOS application ecosystem is smaller compared to the set of embedded ODL applications, since the scope was initially much more narrow. The applications, such as *SDN IP/BGP*, *IP RAN* support for *packet/optical* networks, have been developed for the needs of the service providers. Recently, the two controllers are converging, and ONOS started to offer a support for virtualization of data center networks and interworking with cloud management platforms, such as OpenStack, with the *SONA* project. Arguably the largest ONOS application was the *Central Office Re-architected as a Datacenter (CORD)*, which has evolved into independent open source project. Other notable applications are the *Virtual Private LAN Service (VPLS)* and the *Carrier Ethernet*.

2.2.3 Comparison of ODL and ONOS

The comparison of ODL and ONOS platforms in terms of project maturity, development activity, size (i.e., Lines of Code (LOC)), number of defects and fault density is presented in Table 2.1. The fault density is expressed as the cumulative number of bugs per thousand lines of code. Note that fault density can be expressed accounting only for the bugs reported against the particular software release.

The issues associated to both controllers gathered from the publicly available Jira tracking system, which contain detailed bug reports from the live deployments in both lab and operational environments. The number of detected bugs reported over time are shown in Fig. 2.6. It can be observed that, although the ODL controller has 4.5 times higher bug content than ONOS, the relative bug content, i.e., the fault density, is approximately the same for the two network orchestration platforms.

[7] Adapted from the tutorial presented at the "ONOS Developer Workshop"

[8] Adapted from ONOS documentation: Architecture and Internals Guide - System Components https://wiki.onosproject.org/display/ONOS/System+Components

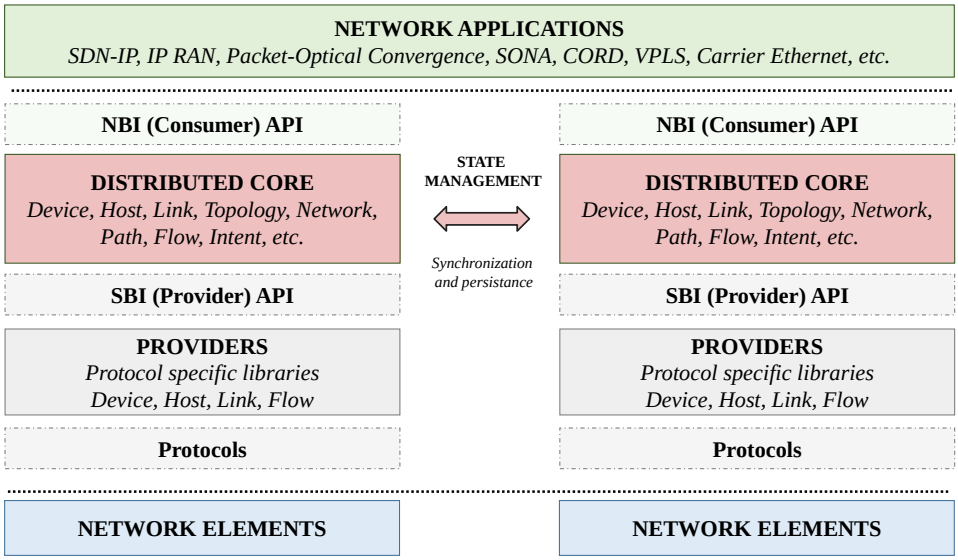


Figure 2.5: Tiers of functionality in ONOS architecture (adapted from "ONOS Developer Workshop").

Table 2.1: Comparison of ODL and ONOS (September 3, 2018).

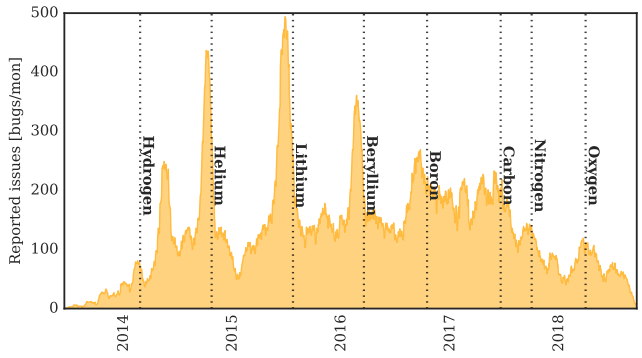
Platform	ODL	ONOS
Project start	February 2013	December 2014
Active release	Fluorine (rel.9)	Quail (rel.17)
Commits	98k	13k
LOC	3,920,556	852,570
Bugs	9,060	2,072
Fault density [$\frac{\text{bug}}{\text{kLOC}}$]	2.31	2.43

2.3 Dependability Assurance in Softwarized Networks

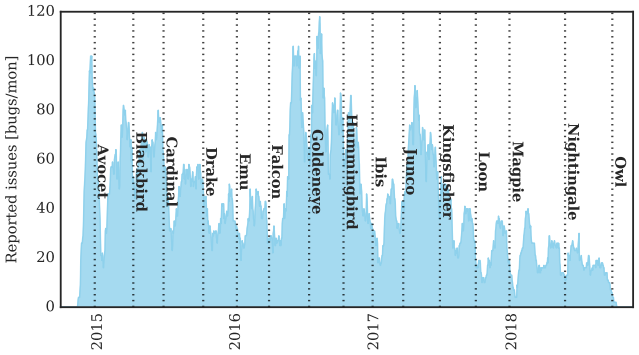
In this section, the SoA on dependability of softwarized networks is presented. The limitations of the previous work are identified (Sec. 2.3.1), followed by an overview of data-driven approaches for software dependability assessment and assurance (Sec. 2.3.2).

2.3.1 Related Work on Dependability of Softwarized Networks

In softwarized networks, the complex network control and management functions are implemented in software, increasing the impact of the software dependability on network performance. The control and management functions, i.e., SDN controller and NFV MANO, are particularly critical for the network dependability, since the failure could disable the services depending of them, e.g., security policy update or provisioning of the new flow paths. Arguably, the most important service is the monitoring of the network infrastructure. E.g., in case of the failure of the SDN controller, both automatic and manual reaction to the network data plane congestion or data plane failures will be disabled, since the operator would not be able to detect and correct them in a timely manner. The characteristics of these failure modes, e.g., triggering events, failure manifestation and failure recovery procedures, have to be carefully studied in order to fully comprehend the impact on the network performance.



(a) ODL



(b) ONOS

Figure 2.6: The number of software defects related to distributed implementations reported over time for ODL and ONOS. The dates of major releases for both distributed controller platforms are indicated in the figure.

Nevertheless, a disproportionate body of work on dependability of softwarized networks was focusing on the hardware failures, in particular link failures, while software failures have been either neglected or oversimplified.

An overview of the representative studies on dependability assurance in SDN and NFV is presented in Table 2.2. The detailed discussion on the SoA is discussed in Chapters 4, 5, and 6. This section summarizes the limitations on dependability assessment and assurance, emphasizing in particular the assumptions made by the related work that are not met by the production grade controllers discussed in the previous section.

Table 2.2: Overview of representative work on dependability assurance in softwarized networks.

Focus	Representative studies
Resilient network design and optimization	Design strategies for robust SDN control plane [72, 100, 157, 163, 15], QoS-aware resource management problems in NFV [46, 153, 13]
Dependability modelling	Modelling SDN control plane availability [60, 121, 127, 128, 161], distributed consensus protocols in SDN (Raft, Paxos) [161, 171], NFV-based virtualized core [61]
Network software design and evaluation	Distributed SDN architectures [28, 70, 97, 119, 148, 176], operational issues in SDN controllers [68, 69, 123, 159–161, 199], performance and scalability assessment [29, 44, 125, 138, 170]

Resilient network design and optimization: The strategies proposed for provisioning of the resilience in SDN data plane, such as resilient routing or critical node detection, are not limited to SDN, and hence, they are omitted from the overview, which focuses on the control plane.

In SDN, the inherently distributed control plane logic of forwarding devices, is extracted and moved to SDN controller. The forwarding devices need a reliable connection to the controller(s), in order for the network to work properly. The first work on the reliable control plane design focused on providing the robust connection to one or more controller replicas, which are placed in the network based on different criteria, e.g., minimizing control plane latency. Different variations of this problem, known as resilient or robust *control placement problem* have been proposed. On another side, the work on the design of reliable NFV networks focused mainly on the provisioning of resilient SFC, which is a problem know as reliable *function placement problem*. The placement of virtual network functions in NFV is very flexible thanks to the fact that software instances can be installed at any general purpose hardware with enough available spare capacity. The function placement has a critical impact on the performance guarantees that operators can provide to their customers, as well as on the cost of the service provisioning. Different failure patterns have been modelled in both cases, from single and double link and/or node failures, to disaster scenarios anticipating multiple geographically correlated failures, such as epidemic failure propagation and centrality based attacks. However, the synchronization overhead of SDN controller replicas and stateful VNFs were not considered.

QoS-aware network design and optimization problems in SDN and NFV based networks have been recognized as important research challenges. The difficulty comes from the fact that they combine several NP-hard problems, such as facility location problem, minimum-cost flow problem, generalized assignment problem, knapsack, etc. Different approaches based on standard techniques for solving such combinatorial optimization problems have been proposed, such as Integer Linear Programming relaxation and rounding, greedy heuristic, meta-heuristic (e.g., Pareto Simulated Annealing, Genetic Algorithms), game theoretic approaches, and more recently machine learning and neural networks. A good overview and a summary of different approaches can be found in [73, 98, 9, 112, 122]. The main drawback of the SoA approaches to QoS-aware design of SDN and NFV networks is that they either ignore completely or offer a very limited support for service availability differentiation. A dedicated protection can guarantee uninterrupted services in case of single link or node failures, but it does not provide explicit guarantees for service availability or downtime distribution, necessary for the proper operation of industrial control networks.

Dependability modelling in softwarized networks: First studies on the reliable network design and optimization consider only control path failures and/or reduce the controller to a single failure mode. More complex failure scenarios including both hardware and software failures, have used different modelling formalisms, from combinatorial (non-state space) reliability models (e.g., Reliability Block Diagrams (RBDs) and Fault Trees (FTs)), to state-space models (e.g., Continuous Time Markov Chains (CTMCs) and Stochastic Reward Nets (SRN)).

An important limitation of the proposed models in SDN is the assumption about the perfect failover between identical controller replicas. The preliminary bug classification presented in Fig. 2.4 shows that the clustering module, responsible for the synchronization of the SDN controller replicas is defective. Moreover, the simple controller replication is ineffective, because of i) shared failures, e.g., semantic bug in path computation, ii) faulty error handling mechanisms, which may lead to a erroneous

failover and cause a cascade of controller failures and iii) failures specific to distributed control plane implementations, such as a software bug in distributed consensus protocols. In system dependability, these inefficiencies are modelled as a *common mode* failure (i), and a *coverage factor* (ii), while the failures specific to distributed systems (iii) are typically neglected. Another important limitation of the proposed models is the assumption about software failure rates, which are assumed to be static, following a negative exponential distribution (Poisson assumption) or Weibull distribution. The bug manifestation rate presented in Fig. 2.6 shows that this assumption does not reflect the stochastic behaviour of real life controllers. Moreover, another limitation of the proposed models is that the interaction with the service and control plane is not considered. Namely, the control plane services are not needed all the time, but only at the times when the new service request arrives (e.g., new flow, policy update, new SFC), for the duration of the request. Consequently, different services will perceive different service availabilities, which must be accurately estimated in case of mission critical scenarios, such as industrial networks.

Network software design and evaluation: The third class of the relevant related work on dependability assessment and assurance in softwarized networks, addresses the implementation and operational issues. One research direction focused on the design of efficient distributed SDN control plane architectures. However, most of the proposed distributed control plane designs remained just as research prototypes, whereas the commercial ones, such as Google's B4 controller, are closed for the research community. ONOS and ODL are the two distributed open source controllers which have been widely deployed in the operational SDN networks. Hence, the focus of this thesis is precisely on them.

The other research direction focused on performance benchmarks; in particular on the performance and scalability assessment, uncovering some unexpected issues, which have not been detected in the testing phase of the software. However, some identified issues, such as the effects of software ageing, have been observed, but not thoroughly investigated. The goal of this thesis is to propose a more systematic dependability assessment frameworks that can be applied for the network software certification for the mission critical systems, such as industrial networks.

2.3.2 Data-driven Software Dependability Assessment and Assurance

This section presents an overview of the relevant studies on qualitative and quantitative analysis of software dependability. While *qualitative* analysis is used to identify the prevalent failure modes and localize vulnerable software components, the *quantitative* analysis leverages the empirical data, using statistical inference techniques and stochastic reliability models, to evaluate and forecast the software dependability attributes.

2.3.2.1 Qualitative Analysis

Qualitative analysis focuses on the nature of the historical failure reports. The documentation on historical bugs and post-mortem reports from the operational environments gives *insights into real-life issues*, helps *identifying prevalent failure modes* and *localizing the most vulnerable system components*.

Large empirical studies on Google [63] and Microsoft operational networks [149] reported that software issues caused more than 30% of customer impacting incidents. Google study [63] further debates that network control plane issues prevail, providing an in-depth analysis of the most critical

control plane defects. Similarly, the analysis of IP Backbone [111] and data center networks [56, 66] provided valuable data to industry and researchers, exposing network vulnerabilities and suggesting preventive measures. First insight into OpenStack bugs were presented in [49]. However, a comprehensive study on network control software in SDN is still missing.

Analysis of the experience reports on similar systems also provides a valuable insight into potential vulnerabilities and reference failure patterns. Relevant empirical studies that inspired the work addressed in this dissertation, analyze the nature of defects in popular open source software [103] and cloud control systems [66], prevalence of resource leaks [53, 54], concurrency issues [107], efficiency of redundancy [48] and testing [198], as well as data center components, both virtual and physical [30, 166].

For instance, Yuan et al. [198] demonstrated how the simple testing of error handling code, can significantly improve the reliability of distributed data-intensive systems, such as Cassandra and Redis. Rahmani et al. [154] found the pattern of bug content evolution in popular open source software, enabling an accurate forecasting of the software reliability, which is further explored in Chapter 4. The efficiency of redundancy in distributed storage systems, discussed in [48], demonstrated that simple component replication does not imply fault tolerance, triggering the study presented in Chapter 5. Ageing related issues in `Java`-based applications and operating systems [37, 40] inspired the analysis in Chapter 6.

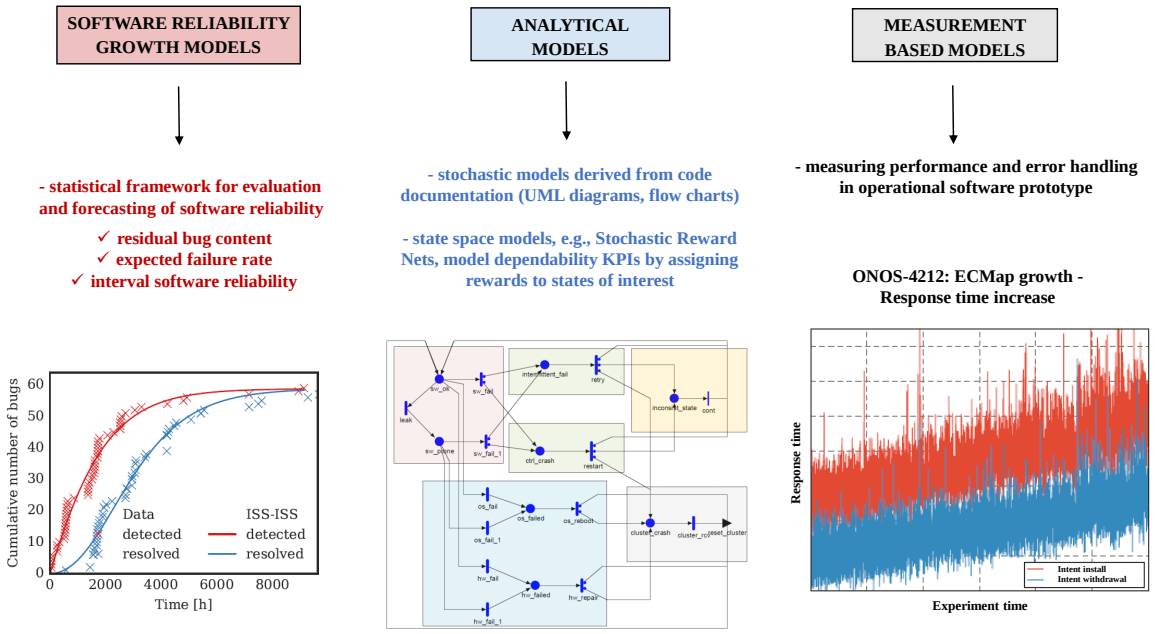
Moreover, such qualitative empirical studies provide a valuable source of information to derive the statistical distributions and parametrize the presented quantitative models. The data source are the public software repositories, code versioning, tracking and documentation systems (e.g., Git, Gerrit, JavaDoc), as well as the public issue trackers (e.g., Jira, Bugzilla). In the case of open source network control software, such as ONOS [137] and ODL [104], the issue trackers contain 10k+ reports about the incidents from test and production environments. These reports include valuable data for data-driven dependability analysis, such as brief bug description, the triggering events and environment in which the bugs manifest, ii) reporting and resolution time, the severity of the issues, indicating an impact the issue had on the system, ranging from minor failures that do not have a significant impact on the system performance to blocker issues.

2.3.2.2 Quantitative Analysis

Next, the data-driven models used for quantitative dependability assessment are presented. These models are using statistical inference techniques and stochastic reliability models, to evaluate and forecast the software dependability attributes, as illustrated in Fig. 2.7.

Software Reliability Growth Models

Software Reliability Growth Models (SRGM) describe the long term improvement in software reliability due to increase of software maturity, through bug removal process. SRGM is a statistical framework, which is often used to estimate the reliability of the software components in their operational phase, based on their fault reports from the testing phase. During the testing and early operational phase of the software lifecycle the faults are detected and removed, which eventually leads to reliability growth (hence the name). A good overview of different classes of SRGMs, their inherent assumptions and input data requirements, can be found in [108, 143]. The SRGMs can be used to predict the



(a) Chapter 4 [5] (b) Chapter 5 [6] (c) Chapter 6 [4]

Figure 2.7: The models for data-driven software dependability assessment and assurance used in this dissertation.

number of *residual bug content*, *expected failure rates*, and *software reliability*. Knowing the statistical distributions of the software failure rates, management KPIs, such as *optimal software release and adoption time* can be determined [78, 95, 99]. Dependability evaluation and forecasting with SRGMs is further elaborated in Chapter 4.

Analytical models

Architecture based models, belonging to the class of analytical white-box models, are derived from the system or the code behaviour specifications, such as flowcharts, UML or sequence diagrams, and are typically specified in the formalism of Reliability Block Diagrams (RBD), Continuous Time Markov Chains (CTMC) and Stochastic Reward Nets (SRN). In the domain of software reliability, three classes of approaches exist: state based, path based and additive models. A good overview and relationship between different architectural models is given by Goseva et al. [62]. Architecture based models model the interactions between different software units, i.e., functions, classes, models and projects. These models are useful in early phase of the software life cycle when the software code architecture can still be modified. This modelling approach can be particularly useful in the case of open source software where the interactions and APIs between different software projects and modules are publicly exposed. SRNs modelling formalism is used in Chapter 5, to model the interaction between SDN controller replicas in distributed implementations, as well as the interactions between control and service planes.

Measurement based approach

Measurement based models focus on empirical evaluation performance and dependability, as well detecting anomalies in operational software. While these models are the most faithful to the real-life system behaviour, they require an operational software prototype to be ready and available for

testing. Measurement based evaluation is also time intensive, many repetitions are required to obtain statistically significant results.

The changes in software reliability due to the effects of software ageing, i.e. gradual degradation of system performance due to the increased resource consumption (e.g. memory bloating and fragmentation, unreleased thread locks), have often been proven and measured empirically. The measurement-based approach is used in Chapter 6 to characterize workload-ageing relationship for critical workload patterns.

Chapter 3

Incentives for Softwarization of Industrial Networks: A Wind Park Case Study

3.1 Introduction

The analysis of technological and economic incentives for softwarization of industrial networks has been analysed in the case study of a wind park. An SDN/NFV-based industrial network prototype deployed in an operational wind park within European project, VirtuWind [110], has provided an insight into operational details of production industrial networks, enabling a realistic assessment of feasibility of softwarized industrial networks.

Wind energy is one of the most affordable and fastest growing sources of renewable energy, with more than 500 GW of installed capacity worldwide in the past 20 years. Incentives from national governments, as well as the ones proposed through the Renewable Energy Directive by European Commission targeting at covering 20% of energy needs with renewables by 2020, further promote the widespread adoption of green power plants. As the number of installed wind parks is rapidly increasing, there is a need for their tighter coupling and the efficient coordination of energy production schedules [187]. Smart Grids, which are considered as a promising solution for the integration of a diverse set of energy production and distribution systems, require deep penetration of ICT technologies in all of its subsystems. However, current wind parks are not yet prepared for a seamless integration into the Smart Grids, mainly due to the lack of mechanisms for automated and secure exchange of information [110].

Industrial communication networks, such as the one in wind parks, which in the past have been developed as closed systems, rely on closed proprietary protocol stacks, that have been tailored and optimized for their particular requirements. Different Industrial Ethernet protocols were developed to accommodate the stringent industrial-grade requirements for latency, jitter and reliability, necessary to provide the stable operation of power control networks. The lack of compatibility between different Industrial Ethernet protocols leads to vendor lock-in, since wind park owners must deploy components from the same manufacturer to ensure their interoperability. Furthermore, the existing wind park communication networks suffer from high configuration and management complexity. Network upgrades and updates are error prone and time consuming as they require customized scripting tools and many hours of testing performed by highly specialized network engineers. Also, network maintenance and

failure reparation are costly and incur a loss of revenues due to reduction of power production, as wind turbine generators need to be taken out of service during the maintenance operations. Moreover, security breaches, such as Ukraine's power plant hack in 2015, are not uncommon, despite the deployment of sophisticated network security appliances. The exposure to cyber-attacks is only expected to increase, in the context of Smart Grids [81].

The 5G concepts of network softwarization, i.e., SDN and NFV, have shown to be a promising solution to solve several practical issues regarding the protocol openness and the fine grained security control, as well as the full automation of network configuration and management [110]. With SDN, the distributed control plane logic of forwarding devices, i.e., switches and routers, is moved to a software entity called SDN controller. The SDN controller provides an integrated interface towards the forwarding devices, which significantly simplifies the network management and augments the network programmability. Providing standardized and open interfaces towards the network components, helps the network operators to avoid the vendor lock-in, and hence to obtain lower prices through the increase of the market competitiveness. In NFV higher layer network devices, such as firewalls or intrusion detection systems, which are traditionally implemented in a specialized hardware, are replaced with modular software components deployed on commodity hardware. Such modular network functions can be further chained to provide fine grained traffic control, offering much greater flexibility and lower cost of the service deployment for wind park network owners and industrial network operators in general.

First studies on software-defined industrial networks have shown that it is possible to achieve deterministic delay [65], high availability and low recovery times [161, 5], and guarantee high security standards [8] with commodity SDN switches and general purpose hardware. The feasibility of achieving industrial grade quality of service with open and extensible protocol suite provided by the SoA SDN and NFV solutions, has been already demonstrated in an operational wind park environment as a part of the VirtuWind project [110]. The goal of this chapter is to discuss and explore the technological and economic incentives for the wind park owners and operators to softwarize their networks. Due to the limited size of the test wind park, which was not representative of commercial wind parks, the presented case study considered a typical off-shore wind parks in Northwestern Europe.

The results presented in this chapter have been published in [8, 18, 19, 12, 7]. The architecture and limitations of legacy wind park communication network have been presented in [19], while the traffic classes and their corresponding QoS requirements have been included in the IETF draft on Deterministic Networking (DetNet) Use Cases [18]. An SDN/NFV-based solution for virtualization and orchestration of security functions for operational wind parks has been presented in [8]. The general framework challenges of design and optimization of dependable softwarized networks for industrial applications are presented in [12], while the techno-economic framework and cost models are elaborated in [19]. The quantitative study on the incentives for softwarization of industrial networks is presented in [7]. The main contributions are the analysis of the requirements of the wind park communication networks, cost models for the analysis of networks' CAPEX and OPEX, as well as the practical case study on the economic incentives for network softwarization. The architecture of the industrial controller prototype is not the part of the author's contributions, and is provided in Fig. 3.4 (Sec. 3.3) only as a reference.

The remainder of the chapter is organized as following. Sec. 3.2 presents an analysis of legacy wind park communication networks. Design and implementation of an industrial network orches-

tration prototype is presented in Sec. 3.3. Sec. 3.4 discusses the techno-economic incentives of the communication network softwarization. The chapter concludes with a summary and discussion of the results, in Sec. 3.5.

3.2 Legacy Industrial Networks: A Wind Park Case Study

This section analyses the architecture and the limitations of the typical closed industrial networks. First, the traffic classes in the wind park communication networks are discussed, as well as the requirements imposed on the underlying network, in terms of data rate, latency, reliability and packet loss. Next, the design and limitations of the existing wind park communication networks are presented.

The principal communication actors in the wind park are located in Wind Turbine Generator (WTG) and Supervisory Control and Data Acquisition (SCADA) system, as illustrated in Fig. 3.1.

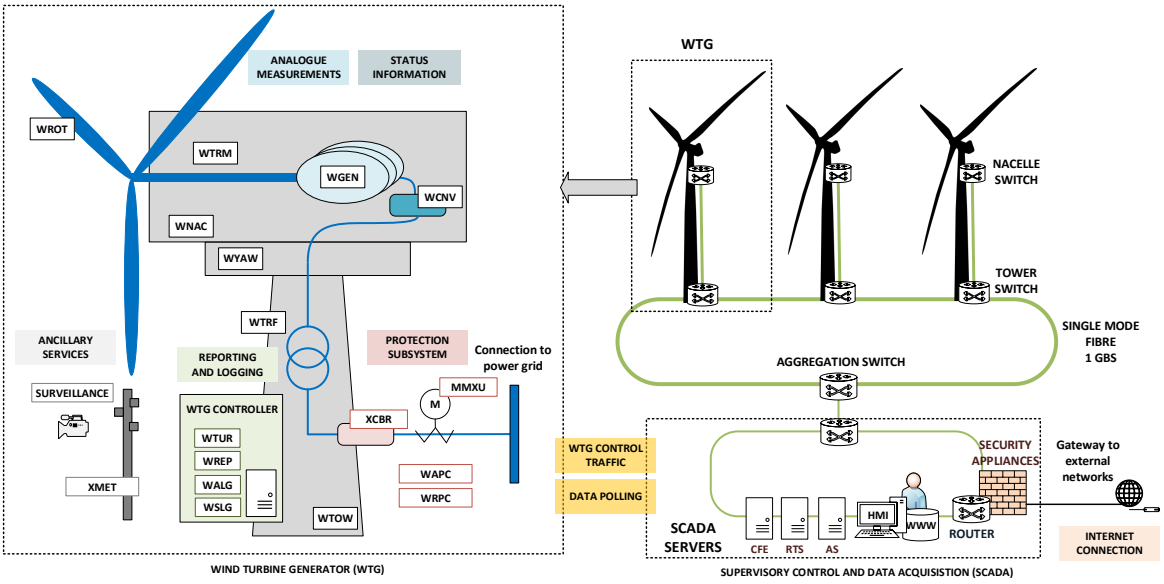


Figure 3.1: Inside the wind park communication network [7] (©2019 IEEE).

3.2.1 Wind Turbine Generator (WTG)

The wind turbine generators represent a complex system of Intelligent Electronic Device (IED) and Remote Terminal Unit (RTU) that consist of sensors, actuators and an internal controller.

According to the international standard IEC 61400-25 "Communications for monitoring and control of wind power plants" [82], which provides a framework for the information exchange within a wind park, IEDs and RTUs in WTGs are grouped into logical nodes based on their function, as shown in Fig. 3.1. Every logical node supports three classes of traffic status information, analogue measurements and control information. For instance, a wind turbine rotor (WROT) sends **status information** regarding the rotor and the blades, **analogue measurements** of the rotor speed and the temperature, and receives the **control information** to set the pitch angle for the blades or set the rotor to a blocked position.

As a part of the substation automation, each WTG must be equipped multiple IEDs or RTUs, such as measurement merging unit and circuit breaker, providing the **protection switching** control against overcurrent and overvoltage. The role of the reporting and logging system is to provide full traceability of sequence of events in case of a failure. It provides information derived from the original measurements and status messages. Reports are provided on demand, while log files are transmitted periodically to the SCADA. Video surveillance is used for security, to detect the ships or vehicles that are approaching the wind park, as well as to monitor the state of the turbines and the environment. Video can be streamed continuously or requested on demand.

3.2.2 Supervisory Control and Data Acquisition (SCADA)

A typical SCADA system, consists of several application servers, as shown in Fig. 3.1. The Communication Front End (CFE) server is used for data acquisition from field devices (RTUs and IEDs), and can also perform protocol conversion and temporary storage of measurements and status data for real-time data trending. The Real Time Servers (RTS) are in charge of data processing, real-time operational process control and short-term data trending, while the Archive Servers (AS) are used for long term data storage. The system also has a Human Machine Interface (HMI) to facilitate user interaction with the network and to allow engineers to access and modify the operational data, and display alarms and power plant status information. The Web Server (WWW) provides an interface to the SCADA for the users that access the system via their personal computer over the Internet.

The traffic from the SCADA towards the wind turbines consist of two components, constant control traffic and periodic data polling. Internet access and interfaces to third party systems are also provided. This includes internal interfaces to meteorological mast and video surveillance, as well as external ones to the Internet, national grid and other control centers, according to IEC 60870[82] and IEEE C37.1-2007[174].

The traffic classes and their QoS requirements are summarized in Table 3.1. The consolidated QoS requirements are based on the relevant industry standards [36, 82, 174] and the previous case studies on wind park [21, 165, 188] and SCADA [145] architectures.

Table 3.1: Traffic classes and services present in the wind park: based on the relevant industry standards [36, 82, 174] and previous case studies on wind park [21, 165, 188] and SCADA [145] architectures (©2019 IEEE).

Service	Priority	Data rate	Latency	Reliability	Packet loss
Protection traffic	1	58 Bytes/second	16 ms	99.999%	$\leq 10^{-9}$
Analogue measurements	2	225,544 Bytes/second	16 ms	99.99%	$\leq 10^{-6}$
Status information	2	76,816 Bytes/second	4 ms	99.99%	$\leq 10^{-6}$
Reporting and logging	3	15 KB every 10 min	1 s	99.99%	$\leq 10^{-6}$
Video surveillance	4	250 Kbps – 1.5 Mbps	1 s	99%	N/A
Data polling	1	100 Bytes/ 2 ms and 2 KB/sec	16 ms	99.99%	$\leq 10^{-9}$
Control traffic	2	20 kbps per turbine	16 ms	99.999%	$\leq 10^{-6}$
Internet connection	3	1 GB every two months	60 min	99%	N/A

3.2.3 Wind Park Communication Network

The communication system in the wind park is designed to guarantee the industrial-grade requirements of the services specified in Table 3.1, required for a reliable flow of control and monitoring traffic between the SCADA and WTGs. WTGs are typically grouped in rings and radials to maximize the energy production. The topology of the communication system is constrained to the layout of the power collection system, since optical fibres that interconnect turbines and the SCADA are embedded in the power line cables. A typical power cable has up to four optical fibres, which support 1:1 protection, and also offer huge capacity to support bandwidth hungry applications, such as video surveillance. The links within a turbine are either optical fibres or twisted pairs. As depicted in Fig. 3.1, there are typically two access switches in each wind turbine: one at the top which is distributing the traffic between sensors and actuators of IEDs and RTUs, wind turbine controller and other ancillary functions and one at the bottom which is handling the traffic between turbines and the SCADA. Core switches aggregate the traffic coming from different radials. Unfortunately, standard Ethernet switches do not provide guaranteed latency since the queuing delay is not bounded. Hence, special switches, implementing Industrial Ethernet protocols, are required to ensure deterministic delay. The ecosystem of switches capable of supporting wind park requirements is rather small. Ensuring the inter-compatibility forces the wind park operators to deploy all network components from the same vendor, such as Connected Grid and Industrial Ethernet switches by Cisco, or complete network solutions provided major wind turbine vendors.

The router and the gateway in the SCADA enable the communication with external networks. Typical wind park routers, such as Cisco 2000 Connected Grid Router, support VLANs with IPSec, which are deployed to isolate different traffic classes and to limit the access to sensitive control traffic only to the authorized users.

Security is of the paramount importance in industrial networks. Advanced security appliances, such as the ones provided by Cisco ASA 5000 Series, comprise of firewall, intrusion detection and prevention system and deep packet inspection functions. Security appliances in legacy wind parks are deployed as software bundles running on the specialized proprietary hardware, which is a setup typically optimized for high volume traffic in data centers and enterprise networks. This approach incurs unnecessarily high cost for the wind park operators. Security breaches are not uncommon, despite the sophisticated mechanisms deployed in the power plants, calling for the design of new security solutions that are tuned better for industrial purposes, such as the one presented in the following section.

3.3 Softwarization of Industrial Networks

Next, the architecture of a softwarized wind park is introduced, discussing how SDN and NFV can be used to solve the practical issues regarding the protocol openness (Sec. 3.3.1), the fine grained security control (Sec. 3.3.2) and highly automated network management (Sec. 3.3.3), as well as an overview of the required network components (Sec. 3.3.4). The architecture of the industrial controller prototype is not the part of the author's contributions, and is provided in Fig. 3.4 (Sec. 3.3) only as a reference.

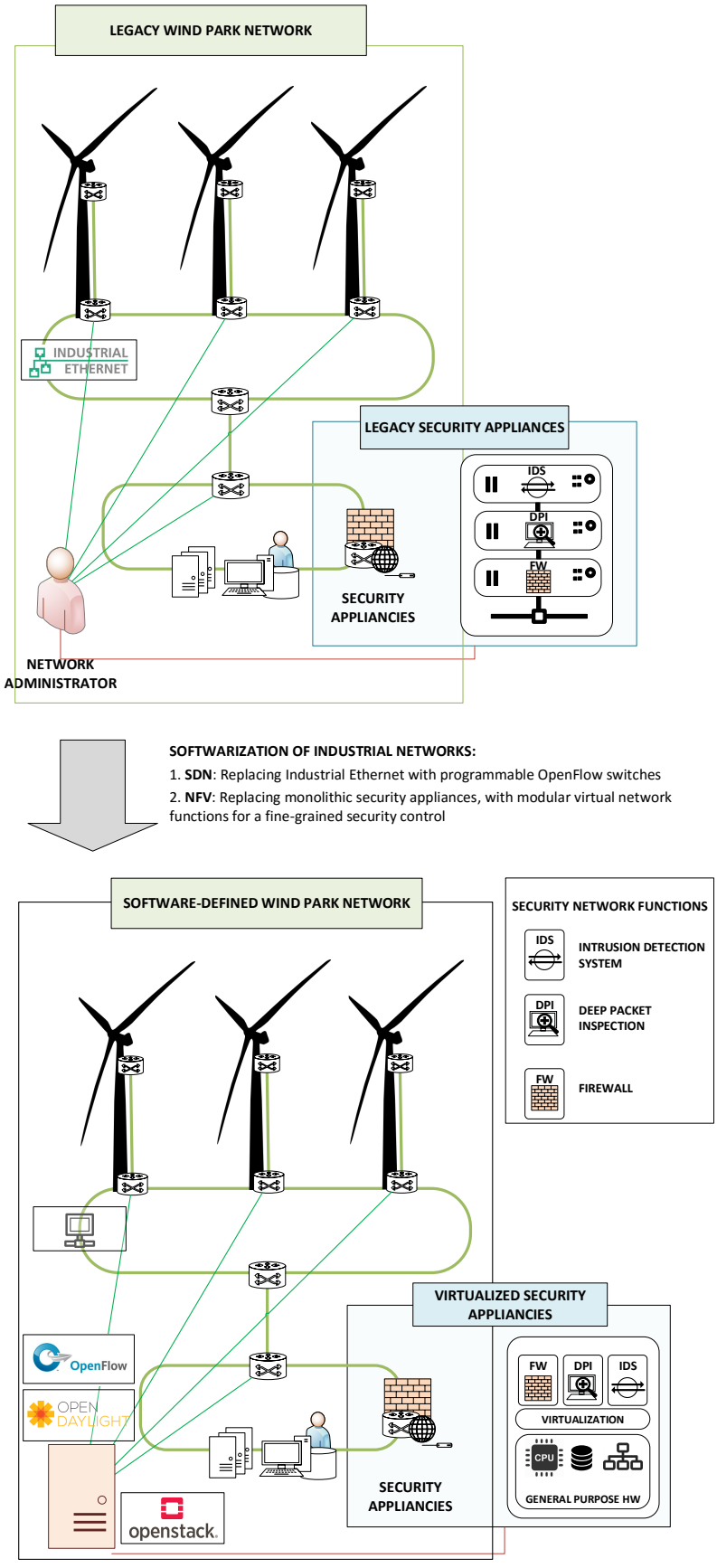


Figure 3.2: Softwarization of industrial networks (adapted from [7]).

3.3.1 SDN: Replacing Industrial Ethernet with Programmable OpenFlow Switches

In legacy wind parks, the industrial grade of service (e.g., deterministic latency) is guaranteed with closed protocol suite based on Industrial Ethernet, since standard Ethernet switches cannot provide bounded queuing delays. In SDN, the switch control plane logic is outsourced to the SDN controller, as illustrated in Fig. 3.2. The controller has a global overview of the network state, and can provide delay guarantees through logically centralized queue-level flow management, proposed by Guck et al. [65].

SDN-enabled switches are simpler, and hence, cheaper than Industrial Ethernet Switches, and the price gap is expected to increase as the technology matures. Already today, there is a myriad of high-end commercial switches already deployed in enterprise and data center networks, from white box solutions (e.g., EdgeCore AS4610 with Pica8) to big networking hardware vendors (e.g., HPE FlexFabric 5930 and Aruba 3800), offered at a competitive price. Moreover, commodity SDN-enabled switches support standard Ethernet, facilitating the seamless integration of different energy production systems, without the need for the protocol converters.

The SDN controller provides high-level network abstraction and vendor agnostic management. This allows the users and applications to specify high level intents, such as opening of a new TCP port at the set of firewalls or the setup of a connection between two hosts with a specified QoS, without minding the low level forwarding rules that need to be configured in the switches. The SDN controller can program the forwarding plane with OpenFlow, an open and standardized protocol managed by the Open Networking Foundation (ONF). Moreover, open source SDN controllers, such as ODL, already provide the support for most of commercial switches.

3.3.2 NFV: Virtualization of Security Network Functions

The most complex network components in wind parks are security appliances, embedding the functionality of firewall, intrusion detection and prevention system and deep packet inspection. Due to the small size of the market for industrial security solutions, wind park operators typically deploy the solutions developed and optimized for enterprise and data center networks. With NFV, the specific security functions can be realized as modular software components running on general purpose hardware, replacing the monolithic security appliances implemented in specialized proprietary hardware, as illustrated in Fig. 3.2. Such setup offers resource pooling, as well as high degree of flexibility when choosing the preferred vendor for the particular security module.

A prototype of an NFV-based solution for industrial networks, based on open source firewall (pfSense) [141], IDS (Snort) [156], deep packet inspection (nDPI) [43] and customized honeypots (HoneyD) [33, 151] has been presented in [8], . These modular software components, which are often packaged as virtual machines, can be further chained to provide a fine grained security control. For instance, unknown traffic flows are first processed and classified by DPI, while trusted SCADA traffic can bypass it to avoid the unnecessary delays. Malicious traffic is redirected to honeypots that emulate wind park network, in order to distract the attackers and allow the operator to collect valuable data about the ongoing attack.

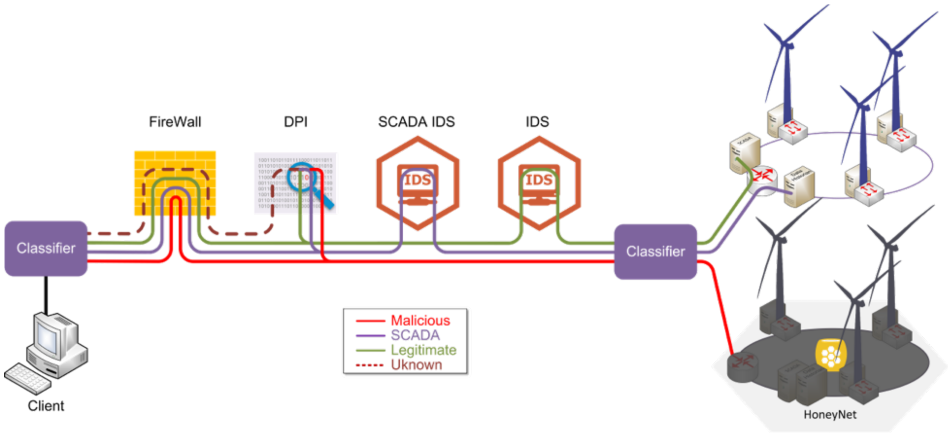


Figure 3.3: Industrial network prototype: virtualized security appliances, from Fysarakis et al. [8] (©2017 IEEE).

3.3.3 Automated Network Orchestration and Management

The recent trend of network softwarization suggest the majority of control and management functions to be implemented in centralized orchestration platforms, i.e., SDN controller and NFV MANO. A VirtuWind prototype of SDN/NFV-based industrial network control platform presented in [162], is illustrated in Fig. 3.4

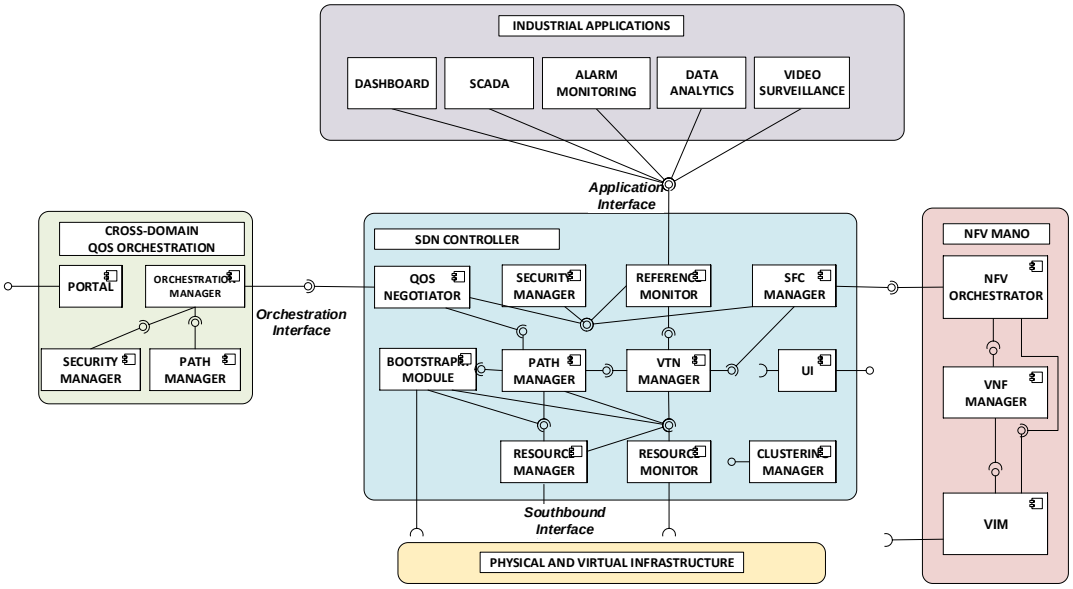


Figure 3.4: Industrial network prototype: control plane architecture, adapted from Sakic et al. [162].

In VirtuWind industrial network platform, the majority of control and management functions are implemented in the SDN controller extending the ODL code base with i) deterministic QoS path provisioning and resource management module [65], ii) adaptive clustering manager [159–161], iii) automated bootstrapping manager, iv) adaptation of ODL's security, VTN and SFC modules, and v) open interfaces for cross-domain QoS orchestration, industrial applications, as well as the management

of physical and virtual infrastructure. NFV MANO utilizes the OpenStack modules for the provisioning and run-time management of modular security VNFs. This approach offers several benefits for the industrial network operator.

SDN: A centrally managed programmable forwarding plane significantly simplifies the setup of new services, since the configuration scripts do not have to be customized for a specific network equipment vendor. Vendor agnostic network control and management are expected to reduce the need for a specialized team of technicians. The automation of network configuration is also expected to reduce the incidence of human errors.

NFV: In legacy networks the customized configuration scripting tools and highly specialized network engineers are required for operation and maintenance of security appliances. On the other hand, NFV offers complete automation of MANO of network functions. The reference architectural framework and MANO interfaces are specified by the ETSI NFV group. Several open source solutions, such as OSM, already provide the solutions for the management of shared physical network infrastructure, virtualization layer and service function chaining.

3.3.4 Industrial Network Prototype Deployed in Operational Wind Park

The overview of the architectural changes introduced by SDN and NFV are illustrated in Fig. 3.2. With SDN Industrial Ethernet switches are replaced by OpenFlow enabled switches, while with NFV monolithic security appliances are replaced by software modules running on general purpose hardware.

Table 3.2: Comparison of the network components in legacy wind park and SDN/NFV-based network: the reference values are the median of the available commercial products from different vendors (©2019 IEEE).

Component	*Network components in legacy wind parks*			*SDN/NFV v.s. legacy network components*		
	Price [€]	Power [W]	MTBF [h]	Price [%]	Power [%]	MTBF^{-1} [%]
Access sw.	3,330	40	263,285	78%	75%	65 %
Aggregation sw.	2,324	100	203,812	87%	95%	65 %
Router and gw.	2,490	210	289,056	85%	67%	90 %
Security app.	3,674	90	299,588	88%	133%	90 %

The comparison of commercial network components in legacy and SDN/NFV-based wind parks is presented in Table 3.2. The network functions implemented in software require additional general purpose servers. The cost of the servers can be divided between all the software components proportionally to their utilization of the physical resources (CPU, RAM, storage). The licensing of the software depends on the business model of the particular vendor in the case of commercial network solutions, while for open source solutions software development and maintenance are provided by the community. Open source network control and management platforms, such as ODL supported by the Linux Foundation, have already shown stable performance in commercial network deployments.

3.4 Incentives for Softwarization of Industrial Networks

In this section, the economic incentives for wind park softwarization are discussed. First, the cost models for capital and operational expenditures are presented, with the goal to quantify the savings that can be achieved by the softwarization of a wind park communication network.The magnitude of savings is demonstrated in the case study of a typical offshore wind park in a Northwestern Europe.

3.4.1 Cost Factors

First, the cost models for CAPEX and OPEX of an industrial communication networks are presented, followed by a case study on industrial communication networks.

3.4.1.1 Capital Expenditures (CAPEX)

CAPEX include all the costs related to the network equipment, including supporting infrastructure and installation cost. Since the goal of the analysis is to evaluate the cost differences between legacy and SDN/NFV-based communication network, this study considers the cost of (i) access switches in WTGs, (ii) aggregation switches, (iii) router and gateway and (iv) security appliances. The cost of the additional blade servers have to be installed in order to support software based network components is also considered.

$$CAPEX = \sum_{i \in Comp} N_i \cdot Price_i \tag{3.1}$$

The number of network components (N_i) that need to be purchased during the lifetime of the wind park depends on several parameters used by network planning such as the component's capacity, the desired redundancy level, and estimated lifetime and vendor warranty period. The traffic volume in wind parks is relatively low (Table 3.1), and active redundancy is typically deployed only in the SCADA. An expected wind park lifetime (T_{oper}) is 20 to 30 years, while the typical lifetime of network components is five to 10 years.

3.4.1.2 Operational Expenditures (OPEX)

Operational expenditures include all the costs related to operation and maintenance of a wind park incurred during the operation and the maintenance of the network. Here, the configuration cost ($Config_{cost}$), power consumption ($Power_{cost}$), preventive maintenance ($Maint_{cost}$), corrective maintenance or failure reparation ($FailRep_{cost}$) and cost of energy not supplied ($CENS$) are considered. All OPEX cost factors are expressed per year, allowing for the comparison of different wind park and industrial networks in general.

$$OPEX = Config_{cost} + Maint_{cost} + Power_{cost} + FailRep_{cost} + CENS \tag{3.2}$$

Configuration cost: Any adjustment of the network, such as the opening of a TCP port or the addition of a new sensor to the wind park network, requires the reconfiguration of network components that has to be performed during the maintenance window, which occurs N_{config} times per year. A team of highly specialized network engineers needs T_{config} man-hours to write and test the configuration scripts. It has been demonstrated that configuration time is significantly reduced in SDN/NFV-based networks, thanks to the high degree of automation and vendor agnostic management provided by the SDN controller and NFV MANO. The hourly cost of the network engineers is w_{nw}.

$$Config_{cost} = N_{config} \cdot T_{oper} \cdot T_{config} \cdot w_{nw} \tag{3.3}$$

Preventive maintenance: Network equipment needs regular maintenance to guarantee acceptable operational conditions as a part of proactive failure management. Inspection of the network equipment is performed N_{maint} times a year and it requires a team of technicians for T_{main} man-hours. The hourly cost of the technicians is w_{tech}. Maintenance activities, such as switch firmware upgrades, are expected to be faster and simplified in softwarized networks since they can be mostly done remotely. In softwarized networks, the network functions implemented in software also require regular maintenance, in terms of feature upgrades and security updates. Primarily the control and management functions, that is, the SDN controller and NFV MANO, need to be updated regularly. Note that described network solution relies on open source components maintained by the community. Even in the case of commercial solutions, the cost of software development, testing and debugging can be shared between all deployed wind parks. Hence, the maintenance cost can be expressed as:

$$Maint_{cost} = T_{oper} \cdot N_{maint} \cdot T_{main} \cdot w_{tech} \tag{3.4}$$

Since the changes in the network configuration are typically executed during the maintenance window, in the remainder of this chapter $N_{config} = N_{maint}$ is assumed.

Power consumption: Given a power cost PC[€/kWh], the power consumption cost can be computed as the sum of the power consumption of all active network components (P_i).

$$Power_{cost} = T_{oper} \cdot 8.76 \cdot PC \cdot \sum_{i \in Comp} P_i \tag{3.5}$$

where a factor of 8.76 accounts for adjustment between units (thousands of hours per year).

It can be seen in Table 3.1 that, while the power consumption of SDN switches and routers is slightly lower than the power consumption of their legacy counterparts, the power consumption of virtualized security appliances running on commodity hardware is actually higher.

Failure reparation The expected number of failures of a network component during its operational lifecycle can be derived from MTBF values provided by the vendors. The repair cost of a single failure depends on the hourly cost of the technicians (w_{tech}) and the time required to repair the failure MTTRi, as well as the cost of their transportation to the site, either SCADA ($Trav_{sc}$) or WTG ($Trav_{wt}$). Note that in the case of offshore wind parks, the time to reach the wind turbine (T_{wt}) is a dominating the actual repair time, and $T_{wt} \gg \text{MTTR}_i$, since a boat or a helicopter may be required for the transportation of technicians.

$$FR_{cost} = T_{oper} \cdot 8.76 \cdot \sum_{i \in Comp} MTTF_i \cdot repair_{cost_i} \tag{3.6}$$

$$repair_{cost_i} = (MTTR_i + 2 \cdot T_{sc/wt}) \cdot w_{tech} + Trav_{sc/wt}$$

Previous case studies have shown that most network outages in legacy wind parks are related to switch port failures. Most of the failures are caused by human error, which is not accounted for in the MTBF values shown in Table 3.2. Since the operation of SDN switches involves minimum human intervention, the reduction of the failure rates, and consequently the cost of failure reparation, is expected to be even higher.

Cost of Energy Not Supplied (CENS): Wind turbine generators need to be taken out of operation during failure reparation. During the interruptions, wind park operators not only lose money that they could have earned by selling the harvested energy, but would also have to pay penalties to the grid operator for not supplying the promised quantity of energy. Given a power penalty PP per interrupted MWh (PP [€/MWh]), expected interruption time IT, a wind turbine power rating (production capacity) WT_0 [MW] and its capacity factor (efficiency of power production) of CF [%], the expected CENS can be evaluated.The interruption time (IT) is larger than the failure reparation time $IT \gg MTTR_i + T_{wt}$ since it also includes failure detection, diagnosis, procurement of the equipment and team preparation.

$$CENS = WT_0 \cdot CF \cdot PP \cdot \sum_{i \in Components} IT_i \tag{3.7}$$

3.4.2 Case Study

The total cost of the ownership of a wind park depends on a number of factors such as the type of project (e.g., number and location of turbines), country specific parameters (e.g., cost of technicians and engineers) and network design parameters (e.g., aggregation factor). In order to illustrate the magnitude of savings due to network softwarization, the case study of the typical offshore wind park in Northwestern Europe is presented. The relevant case study parameters are summarized in Table 3.3.

Table 3.3: Case study: typical offshore wind park in Northwestern Europe (©2019 IEEE).

Wind park parameters		Country specific parameters		Network specific parameters	
Oper. time	$T_{oper} = 20yr$	Power consum.	$T_{oper} = 0.28 \frac{€}{kWh}$	Maint. freq.	$N_{main} = 4$
Size	$N_{wt} = 80$	Technician cost	$w_{tech} = 58 \frac{€}{h}$	Config. (old)	$T_{conf} = 8$h
Power rating	$WT_0 = 4$ MW	Nw. eng. cost	$w_{nw} = 52 \frac{€}{h}$	Config. (new)	$T^*_{conf} = 15$min
Capacity fact.	$CF = 40\%$	Travel (SCADA)	$Trav_{sc} = 100€$	Maint. (old)	$T_{main} = 8$h
Travel time	$T_{wt} = 24$h	Travel (WTG)	$Trav_{wt} = 1000€$	Maint. (new)	$T^*_{main} = 30$min
Interrupt. time	$IT = 120$h	Power penalty	$PP = 150 \frac{€}{MWh}$	Aggregation	$AG = 8$

The contribution of the individual CAPEX and OPEX cost factors is presented in Fig. 3. Significant savings can be observed in both CAPEX and OPEX. More than 442,000€, that is, 19.85 %, of savings can be achieved in CAPEX thanks to the lower cost of softwarized network components. OPEX reduction is expected to be even higher, around 34 percent, accounting for more than 1,380,000€ accumulated savings during the lifetime of the wind park. The reduction of the cost of the access switches in the wind turbine contributes most to the CAPEX savings. The highest cost reduction in OPEX is expected from CENS, due to the shorter interruptions of power production. The second biggest contribution to the OPEX savings comes from failure reparation, due to the significantly lower failure rates.

Provided that some of the baseline scenario parameters have high uncertainty, as well as the fluctuations due to the regional differences, the local sensitivity analysis has been conducted to estimate the impact of individual factors on the total savings. As expected, the number of turbines and the lifetime of the wind park have the highest impact, since it influences all cost components. The factors driving CENS (CF, PP, IT, WT_0) and failure reparation ($w_{tech}, Trav_{wt}$) also have a significant impact.

The impact of the wind park size on the expected savings has been investigated. In large wind parks with more than 300 wind turbines (e.g., the Hornsea in the U.K. has 342 turbines), the total savings

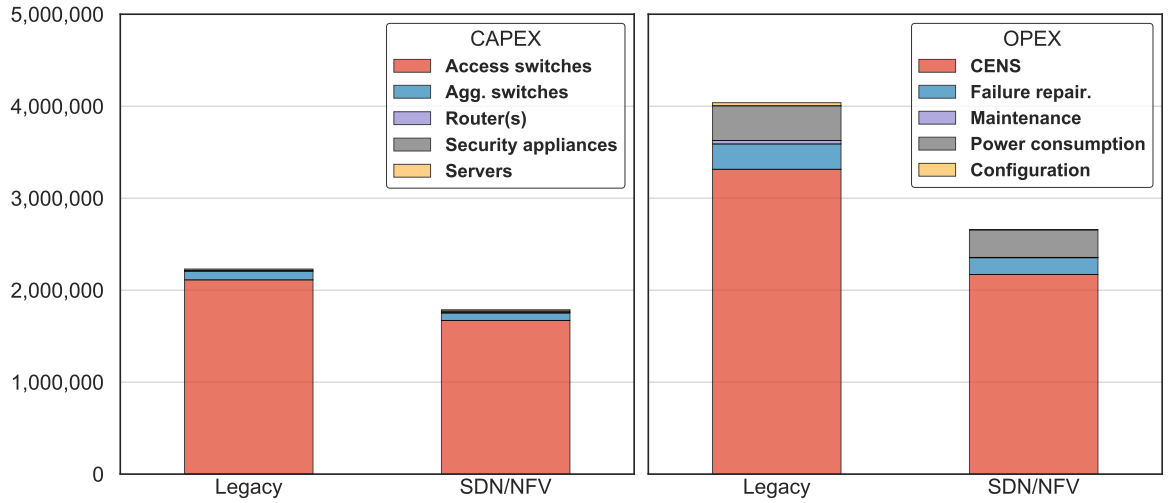


Figure 3.5: Analysis of economic incentives for softwarization of the wind park: 19% of the savings in CAPEX and 34% in OPEX can be expected (©2019 IEEE).

are estimated to be more than 7 Mil. €. The relative savings, however, do not change significantly with respect the wind park size, and it converges to 20 percent of CAPEX and 35 % of OPEX savings in communication network cost.

3.5 Concluding Remarks

3.5.1 Summary

In this chapter, a study of the techno-economic feasibility of the softwarization of wind park communication networks has been presented. SDN and NFV are introduced to solve the limitations of legacy wind parks by providing the protocol openness and the fine grained security control, necessary for the tighter integration of wind parks into future Smart Grids. The capital and operational expenditures have been modelled in order to quantitatively evaluate the benefits of SDN and NFV. A case study of a typical wind park in North-western Europe has demonstrated that significant savings can be achieved through network softwarization, making it a promising solution to facilitate its seamless integration into the Smart Grids. The advantages of network softwarization in wind parks trigger new open questions for the operators such as the identification of seamless migration paths while guaranteeing simultaneous park operation.

3.5.2 Discussion

Threats to validity. The main threat to validity of the presented analysis for the incentives for softwarization of industrial networks is the uncertainty regarding the input cost structure parameters. While technological benefits have been verified and confirmed by VirtuWind industrial partners, the exact cost of the network infrastructure, operation and maintenance operations is not available, due to the privacy reasons. Nevertheless, the estimated cost structure is based on the reasonable assumptions

derived from the public studies and public vendor equipment data, and hence represents a realistic estimation of the magnitude of savings that can be expected by industrial network operators.

Generalization of the results. The industrial SDN/NFV-based network prototype, enabling the deterministic Ethernet, fine-grained security control and automated network management and orchestration has an application in diverse industry verticals. Furthermore, the analysis of incentives for softwarization of industrial networks presented in this chapter can be extended to other local industrial networks, with static communication actors, such as sensors and actuators in the wind parks. Although the actual cost structure can very in different use cases, the framework to assess the expected CAPEX and OPEX savings, similar to the one proposed in this chapter can be applied.

Future work. As a part of the future work, the analysis of the incentives for softwarization of industrial communication networks be extended to support more complex scenarios, such as enabling of massive machine type communication in Wireless Sensor Networks (WSN), and QoS orchestration of geo-distributed industrial networks, e.g., Smart Grids.

Enabling massive machine-to-machine communications in software-defined WSN. The scale and mobility of the actors in wireless sensor networks introduce a new set of challenges for end-to-end dependability assurance. Interference between the actors competing for the limited wireless channel resources and end-to-end dependability assurance in hybrid wired-wireless setups have been studied in [67, 184, 202]. While softwarization of legacy WSNs can alleviate the interference coordination, it increases the complexity end-to-end dependability assurance, especially in setup where control plane traffic relies on the noisy wireless links, which needs to be carefully studied.

QoS-aware orchestration geo-distributed industrial network slices. The communication between geo-distributed industrial sites, such as Smart Grids and IoT edge clouds [51], typically operates over the commercial WANs. The network operators provide probabilistic guarantees on minimum guaranteed bit rate, maximum delay and connection availability, formalized in terms of Service Level Agreements (SLAs). The existing work focused on bit rate and delay, while service availability in the context of wide area networks received little attention. An interesting research direction is design of adaptive service availability-aware resource management strategies for the fair coexistence of industrial and human-centric traffic [11–13].

Chapter 4

Assessing the Software Maturity with Reliability Growth Models

4.1 Introduction

4.1.1 Motivation, Problem Scope and Research Challenges

The recent trend of network softwarization suggests a radical shift in the implementation traditional network intelligence, decoupling network component functionality from underlying hardware. For instance, in SDN, all critical control plane functions are outsourced to an SDN controller. The SDN controller assumes the role of the network operating system, providing an integrated interface towards the forwarding devices, i.e., switches and routers, which significantly simplifies the network management and augments its programmability [86]. The controller monitors the state of the network by gathering the statistics from forwarding devices, makes the global routing decisions, and reacts on the events, such as link congestion or switch failure. In order to fulfil the long list of tasks, today's production-grade controllers, such as ONOS [137] and ODL [104], have become complex pieces of software, consisting of more than a million lines of code. Such a large and complex[1] software inevitably contains bugs, that may disrupt the network operation and corrupt its performance. A study on the hazards in Google's network infrastructure [63], which is partially based on SDN, reported that software bugs contributed to more than 33% of the high impact failures documented in post-mortem reports. Another large-scale study by Microsoft [105] on root causes of customer-impacting incidents in their production networks reports similar results, and shows that software bugs contributed to 36% of critical outages, more than hardware failures and human errors.

Despite the magnitude and ubiquity of software failures, there is a lack of tools to quantify software maturity, and predict the risk of the software related outages in SDN. The performance reports and benchmarks on SDN controllers are still limited to scalability and latency related metrics, such as flow burst install throughput or flow reroute latency. The reliability of the controller software, which is still a big concern and a major obstacle for the wide spread adoption of SDN in commercial telecom and industrial networks [183], is addressed only by a limited number of studies [161, 14, 17].

[1] As a reference, the latest Linux kernel has around 20 million lines of code.

4.1.2 Methodology: Software Reliability Growth Models (SRGMs)

SRGM is a statistical framework, used to estimate the reliability of software components in their operational phase, based on the bug manifestation reports from the testing phase. During the testing and early operational phase of the software lifecycle, faults are detected and removed, which eventually leads to the reliability growth. The core idea behind SRGM is to describe the fault detection and fault resolution as stochastic processes, whose parameters can be estimated from the empirical data, i.e., the history of the previous bug manifestations. Once the best stochastic model is found and parametrized, it can be used to estimate reliability metrics, such as *residual bug content*, *failure intensity*[2] or *expected time until the next failure*, and conditional *software reliability*. Moreover, SRGMs enable operators to estimate how the reliability metrics change over time, as the software matures. Such software reliability metrics capture the relationship between the testing effort and the software quality, which is highly relevant for the software developers of SDN controllers. With SRGM, the risk of the software outages in a given period of time can be predicted with high accuracy, providing the guidelines for the operators of SDN-based networks to take the calculated risk and estimate the best software adoption time, according to the reliability requirements of their network applications. The practical value of SRGM was recognized by the NFV community, which has already included it in the guidelines for the assessment of end-to-end reliability [45].

4.1.3 Key Contributions

The main goal of the proposed framework for assessment of the software maturity is to provide means to evaluate and forecast the reliability of network control software, for both developers and network operators. The workflow steps: data collection, model selection, evaluation of reliability and management Key Performance Indicatorss (KPIs), are illustrated in Fig. 4.1, highlighting our main contributions:

i. Data collection: The empirical data, i.e., the cumulative number of detected and resolved bugs, is gathered from the public issue trackers. High-level descriptive dataset statistics, e.g., distributions of time between failures and time to resolve bugs, is also presented.

ii. Model selection: The best SRGM to describe the stochastic behaviour of the bug detection and the bug resolution processes in SDN controllers is selected and parametrized. The analysis has shown that the bug detection process can be accurately modelled with the class of 3-parameter S-shaped SRGMs. A new class of SRGMs for fault correction process is proposed, as well as their corresponding fitting technique.

iii. Evaluation of reliability and management KPIs: Several applications of SRGM for the management of the quality of network control software evolution over time are presented, e.g., the guidelines for the *optimal software release and adoption time*. Furthermore, two novel applications relevant for the SDN community are proposed: i) *early prediction of software reliability* using transfer learning, i.e., extrapolation of the stochastic behaviour of the previous releases and ii) *the software maturity* metrics as a comparison criteria between alternative software solutions.

[2]The terms bug (manifestation) and failure are used interchangeably in the context of software dependability.

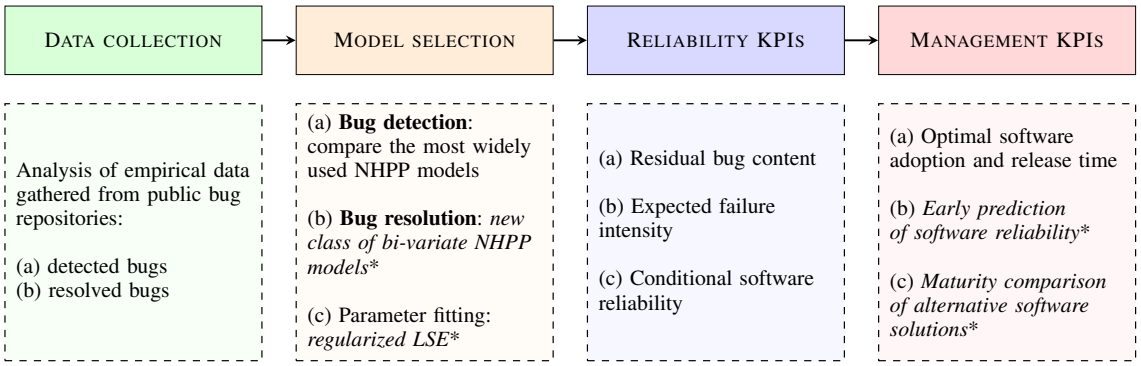


Figure 4.1: Assessment of software maturity with Software Reliability Growth Models (SRGM) consists of four steps: (i) data collection, (ii) model selection, (iii) evaluation of reliability KPIs and (iv) management KPIs. The workflow enhancements proposed in the thesis are marked (*).

The results presented in this chapter have been published in the peer-reviewed conference [17] and journal [5]. The first results [17] focused on applicability of SRGM to ONOS open source controller. In [5] the results were extended to ODL controller platform, exploring different applications of the proposed framework for an efficient management of SDN-based networks.

The rest of the chapter is organized as following. In Sec. 4.2 the limitations of the related work are discussed, while in Sec. 4.3 a theoretical background on SRGM is presented. The case study on open source SDN controller platforms is presented in the following sections, empirical dataset collection (Sec. 4.4), model selection (Sec. 4.5) and network management applications (Sec. 4.6). Summary and discussion of the results are presented in Sec. 4.7.

4.2 Related Work

4.2.1 Stochastic Models for Software Reliability in SDN

In this section, the limitations of the related work on stochastic models for software reliability in SDN are discussed, highlighting the assumptions that possibly contradict the behaviour of the real production grade controllers.

The first studies on the reliability of SDN control plane considered the controller as perfectly reliable, assuming only control path link failures [72]. More recent studies made different assumptions about the controller reliability [60, 106, 127, 131, 157, 161, 15]. Some assumptions were over-simplifying the nature of failures, as they were necessary to obtain analytically tractable results, rather than reflecting controller behaviour from real life deployments or testbed measurements. The authors in [15] modelled controller reliability as deterministic variable. Several other studies [127, 131, 161] assumed that the controller failure to be a Poisson process, which was necessary to obtain analytical solutions of the proposed Markov models. Ros et al. [157] assumed that the operational probabilities of network elements, including the controllers, follow different i.i.d. Weibull distributions. Longo et al. [106] discussed the limitations of Markovian models, and assumed the reliability of the controller to follow phase-type distribution (generalized hypoexponential distribution), which captures better the changes in operational conditions. In the author's preliminary work [14] the instantaneous availability

of the controller software was modelled by hyperexponential distribution, which models different failure types (i.e., transient, hang and crash). The model also included temporal fluctuations of controller software failure rates, which change due to maturity release and state of the controller software instance. This work focuses on the long term variations of software reliability, i.e., the software maturity, based on data from real life production grade open source controllers, ONOS and ODL, demonstrating that the stochastic behaviour of bug manifestation and removal processes can be described with SRGMs.

4.2.2 Reliability Modelling, Evaluation and Forecasting with SRGM

SRGMs have been widely used to estimate and predict the reliability of the software, and in the past, many different models have been proposed, out of which the Non-Homogeneous Poisson Process (NHPP) class has received the most attention. A comprehensive overview of different classes of reliability growth models, their inherent assumptions and input data requirements, can be found in [108]. In this section, the most relevant models, methods and tools for the fitting of model parameters, as well as the applications of the software reliability assessment are discussed.

Models for bug detection: The applicability of SRGMs for the modelling, analysis and evaluation of software reliability of open source products was demonstrated in several case studies. Zhou et al. [201] showed that the Weibull distribution can describe well the bug manifestation process for eight unnamed software projects. Rahmani et al. [154] confirmed this result by analyzing the bug reports for several popular big open source projects, such as Apache HTTP server, Eclipse IDE and Mozilla Firefox. Rossi et al. [158] studied failure occurrence pattern across different releases of Mozilla Firefox, OpenSuse and OpenOffice.org. All studied releases showed the learning curve pattern, where the fault detection rate is slow at the beginning until the community gets familiar with the product, then it increases rapidly until only very few faults, whose discovery is difficult, remain in the code. This effect is captured well with S-shaped models. Syed et al. [175] and Ullah et al. [180] studied the difference between closed and open source software with inconclusive results. In this chapter, eight most widely used SRGMs for the fault detection process [57, 59, 79, 126, 133, 134, 192, 193] are compared in terms of their ability to describe the empirical data of open source network control software.

Models for bug removal: The majority of the SRGMs assume that once a bug is detected, it is corrected immediately, that the debugging as always successful and it does not introduce new bugs. A number of studies have modelled different aspects of imperfect debugging [58, 77, 91, 135, 146, 189]. Wu et al. [189] described the fault resolution as a delayed fault detection process; Pham et al. modelled the introduction of the new faults [146], while Huang et al. [77] also include the changes in debugging effort. Kapur et al. [91] proposed a unified approach to model the fault resolution process, when both fault detection and fault removal are NHPP. Gokhale et al. [58] applied the Non-Homogeneous Continuous Time Markov Chain (NH-CTMC) to model the impact of arbitrary debugging policy, while the study by Okamura and Dohi [135] modelled the time dependency between the bug detection and removal processes as a correlation. These models have a large number of parameters that have to be estimated, while the number of data samples in the historical reports is often very limited (as in the case with ONOS SDN controller), which increases the risk of overfitting the data, as well as the sensitivity of parameter fitting to the noise in the data. In order to balance between the model accuracy and generalizability, a simpler class of models, based on the framework presented in [91] is proposed.

Model parameter fitting: The statistical inference techniques to estimate the parameters of SRGM are Maximum Likelihood Estimation (MLE) and Least Square Estimation (LSE), while historically Method of Moments (MoM), graphical and simulation based approaches have been used [108]. While MLE is convenient for estimating the confidence intervals, LSE is faster and easier to apply to the regularized models described in the following section. Fitting of the model parameters to the empirical data is done either with proprietary general purpose statistical packages, such as SPSS, or specialized tools, such as CASRE [109], SREPT [178] and CARATS [35], just to name the few. In order to account for the newly proposed models, and enhancements in the parameter fitting procedure, a custom tool has been developed based on the libraries provided by the Python scientific package [89].

Applications of software reliability assessment: Software reliability metrics derived from SRGMs, such as expected residual bug content, can be used to balance the trade-off between the cost of software testing and the software maintenance phase, which is known as the optimal software release problem. Since the first study by Okumoto and Goel [136], many researches have analyzed the optimal software release problem under different constraints [78, 90, 95, 96, 99, 194, 195]. Koch et al. [96] provided a cost-benefit analysis for releasing the software after the scheduled deadline, while Yamada et al. [194] proposed optimal software release policies minimizing the total expected cost, under minimum reliability requirements. The authors in [195] considered the optimization of the test-effort allocation to different software modules under the constrained budget for the testing expenditures, while Huang et al. [78] analyzed the impact of different test effort allocation strategies. Kimura et al. [95] considered different software maintenance models, i.e. warranty policies. Lai et al. [99] extended the cost model to capture the additional effort of documentation and distribution of the software patches. In this chapter, two novel applications of SRGMs for the management of software quality are proposed, namely i) early prediction of software reliability based on the transfer learning, i.e., extrapolation of the stochastic behaviour of previous releases and ii) software maturity metrics as a comparison criteria between the alternative software solutions.

4.3 Software Reliability Growth Models

In this section, a theoretical background on SRGM is presented, with the focus on a particular class of models that describe the bug detection and resolution process as NHPP, that have been very successful in modelling the behaviour of large open source projects. The models for the bug detection process, presented in Sec. 4.3.1, are well known models in software reliability community. The composite models for the bug removal process described in Sec. 4.3.2 are novel and extend the existing SRGM literature.

4.3.1 Bug Detection Process as NHPP

Assume that the initial bug content, i.e., the number of bugs present in the software before the start of the testing phase, is a random variable N_0 following the Poisson PMF with the mean a:

$$P(N_0 = n) = \frac{a^n}{n!} e^{-a} \tag{4.1}$$

The probability of detecting a single bug by the time t follows an arbitrary distribution $F_d(t)$. Assuming that the bug detection times are independent and identically distributed random variables,

the conditional PMF of number of detected bugs N_d by the time t is:

$$P(N_d(t) = k|N_0 = n) = \binom{n}{k} F_d(t)^k (1 - F_d(t))^{n-k} \tag{4.2}$$

The unconditional probability of observing exactly k bugs by the time t is then described by the equation:

$$\begin{aligned} P(N_d(t) = k) &= \sum_{n=k}^{\infty} P(N_d(t) = k|N_0 = n) P(N_0 = n) \\ &= \frac{[aF_d(t)]^k}{k!} e^{-aF_d(t)} \end{aligned} \tag{4.3}$$

The process is fully described by the mean value function $m(t)$, which represents the expected number of detected faults by the time t:

$$E[N_d(t)] = m(t) = aF_d(t) \tag{4.4}$$

From the mean value function of the bug detection process, several reliability metrics of the software can be estimated. The instantaneous bug manifestation, i.e., bug detection rate is:

$$\lambda(t) = \frac{\mathrm{d}\, m(t)}{\mathrm{d}\, t} = a\, f_d(t) \tag{4.5}$$

Assuming a finite number of initially introduced bugs $\lim_{t\to\infty} m(t) = a$, the expected number of the undetected faults in the software, i.e., the residual bug content, is defined as:

$$r(t) = E[a - N_d(t)] = a - m(t) \tag{4.6}$$

The conditional software reliability is defined as the probability of not detecting a new bug in the time interval $(t, t + x]$:

$$R(x|t) = e^{-\int_t^{t+x} \lambda(x)\,\mathrm{d}x} = e^{m(t)-m(x+t)} \tag{4.7}$$

The expected cost of the software consists of the cost of testing $c_t(t)$ in the pre-release phase, and the cost of removing the fault $c_w(t)$ in the operational phase during the warranty period T_w of the software lifecycle.

Assuming that the software is released after T time units of testing, the total cost of software maintenance is:

$$C(T) = \int_{t=0}^{T} c_t(t)\mathrm{d}t + \int_{t=T}^{T+T_w} c_w(t)\lambda(t)\mathrm{d}t \tag{4.8}$$

The eight most widely used NHPP models for modelling of the fault detection process are: Musa-Logarithmic, Goel-Okumoto Exponential, Generalized Goel-Okumoto, Inflection S-shaped, Delayed S-Shaped, Yamada-Exponential, Gompertz and Logistic. Their mean value function $m(t)$ and failure intensity $\lambda(t)$ are given in the Table 4.1. The shortlisted NHPP models in Table 4.1 represent well the space of the possible software reliability growth patterns: containing a) three concave and five S-shaped models, as well as b) seven finite failure models and one infinite failure model.

Table 4.1: Fault detection process as Non-Homogeneous Poisson Process (NHPP)

Model	Abbreviation	Shape	$m(t)$	$\lambda(t)$
Musa-Okumoto logarithmic [126]	MUSA(Log)	Concave	$a\ln(1+bt)$	$\frac{ab}{1+bt}$
Goel-Okumoto exponential [57]	GO(Exp)	Concave	$a(1-e^{-bt})$	abe^{-bt}
Generalized Goel-Okumoto [108]	GGO	S-shaped	$a(1-e^{-bt^c})$	$abct^{c-1}e^{-bt^c}$
Ohba's inflection S-shaped [133]	ISS	S-shaped	$a\frac{1-e^{-bt}}{1+\phi e^{-bt}}$	$abe^{-bt}\frac{1+\phi}{(1+\phi e^{-bt})^2}$
Yamada delayed S-shaped [192]	DSS	S-shaped	$a(1-(1+bt)e^{-bt})$	ab^2te^{-bt}
Yamada exponential [193]	YEX	Concave	$a(1-e^{-r(1-e^{-bt})})$	$abre^{-bt}e^{-r(1-e^{-bt})}$
Gompertz [134]	GOMP	S-shaped	ak^{b^t}	$a\ln b\ln k\, b^t\, k^{b^t}$
Logistic[79, 108, 192]	LOGIST	S-shaped	$\frac{a}{1+ke^{-bt}}$	$\frac{abke^{-bt}}{(1+ke^{-bt})^2}$

4.3.2 Bug Resolution Process as Bi-variate NHPP

The bug resolution process consists of two phases, fault detection and correction. Assuming two processes in these two phases are independent, the density function of resulting fault resolution time can be expressed as [189]:

$$f_r(t) = \int_{x=0}^{t} f_d(t-x) f_c(x) \mathrm{d}\, x = [f_d * f_c](t) \tag{4.9}$$

where $f_d(t)$ and $f_c(t)$ represent the densities of the fault detection and fault correction process, respectively. The mean value function of the resulting fault resolution process is then defined as:

$$m_r(t) = aF_r(t) = a\int_{\tau=0}^{t} [f_d * f_c](\tau) \mathrm{d}\,\tau \tag{4.10}$$

Eq. (4.10) can be used to generate different SRGMs from arbitrary distributions for the fault resolution process. However, the proposed models so far have been limited to the combinations for which this integral has a closed form solution, e.g., when both fault detection and correction are Goel-Okumoto processes [91, 189].

$$m_r^{go-go}(t) = a\left[1 - \frac{b_1 e^{-b_2 t} - b_2 e^{-b_1 t}}{b_1 - b_2}\right] \tag{4.11}$$

By replacing the integral in Eq. (4.10) with its Piecewise Constant Approximation (PCA), a numerical approximation for an arbitrary combination of NHPP models can be obtained, which then can be used for the fitting of the fault report data.

$$F_r(t) = \lim_{\Delta x \to 0} \sum_{i=0}^{n=t/\Delta x} [f_d * f_c](i\Delta x)\Delta x \tag{4.12}$$

In this thesis, the four combinations of Generalized Goel-Okumoto and Inflection S-shaped models for fault resolution process are compared, which were preselected due to their performance. The combined Goel-Okumoto Eq.(4.11) from [91] is used as a reference.

4.3.3 Fitting of the model parameters

The LSE method, which minimizes the squared distance between the observed and expected data, is used for the fitting of the model parameters. Unconstrained problems in the model selection phase (Sec. 4.5), are solved using the Levenberg-Marquardt (LM) algorithm. Sec. 4.6.2 provides the bounds on the model parameters, based on the observed parameter trends in the previous releases. The regularized model is solved using the Trusted Region Reflective (TRF) algorithm. Implementation of both methods is provided by Python scientific computing package SciPy [89].

Three Goodness of Fit (GoF) metrics are used to evaluate the suitability of the models: Mean Square Error (MSE), Theil's statistics (TS) and coefficient of determination (R^2), defined as following:

$$MSE = \frac{1}{k}\sum_{i=1}^{k}(m(t_i) - m_{est}(t_i))^2 \tag{4.13}$$

$$TS = \sqrt{\frac{\sum_{i=1}^{k}(m(t_i) - m_{est}(t_i))^2}{\sum_{i=1}^{k} m(t_i)^2}} * 100\% \tag{4.14}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{k}(m(t_i) - m_{est}(t_i))^2}{\sum_{i=1}^{k}(m(t_i) - \overline{m})^2} \tag{4.15}$$

where $m(t_i)$ represents the observed data, and $m_{est}(t_i)$ the data estimated by the model, at time instance t_i of the i-th bug report, and $\overline{m} = \frac{1}{k}\sum_{i=1}^{k} m(t_i)$.

MSE is used to select the best model for individual releases, while TS is more suitable to compare the GoF across different software releases. R^2 is used to measure which portion of variance in data can be explained by the model.

4.4 Data Collection and Preprocessing

The analysis of the software reliability described in the previous section requires complete and uncensored bug reports, which are publicly available only for the open source controllers. At present, there are only two production-grade open source SDN controller platforms, ONOS [137] and ODL [104]. The overview of the architecture of these controller platforms has been presented in Sec. 2.2. In this section, the release and bug management system of the two controllers are presented and compared. The data set preparation, i.e., data gathering and filtering, is described, as well as the preliminary data set analysis, providing the first insight of its statistical behaviour. Descriptive statistics metrics are presented, such as the monthly bug report rates, and distribution of their equivalent Time to Fail (TTF) and Time to Repair (TTR). The data set analyzed in presented throughout this chapter was retrieved on February 1, 2018 from Jira issue trackers of ONOS and ODL.

4.4.1 ONOS Dataset

ONOS is a carrier-grade controller, aiming to fulfil design requirements of large operator networks. New releases are distributed every quarter, ensuring a steady feature development through incremental upgrades of the code base. The three-month release lifecycle starts with a release planning meeting,

followed by three months of code development and integration on the master branch. The feature integration is stopped two weeks before the official release date, and only bug fixes are allowed. The support, including security patches and fix for the critical defects, is provided for the six months after the official release date. Thirteen releases have been distributed between December 2014, when ONOS code was first opened to the public, up to data set collection (February 2018). The releases are named by the bird species, in alphabetical order.

The issues associated to every release are reported in the publicly available Jira tracking system. The issues labelled as "Bugs", rather than new feature requests or enhancements, are filtered out. The bug repositories contain the detailed fault reports from the live deployments in both lab and operational environments. The bug reports contain the information details such as affected versions, bug description and short summary, priority, and report creation and resolution dates.

The cumulative number of detected and resolved faults reported over time are shown in Fig. 4.2a. It can be observed from the figure that there is a steady increase in the number of bugs, with the trend changes being noticeable around the official release dates.

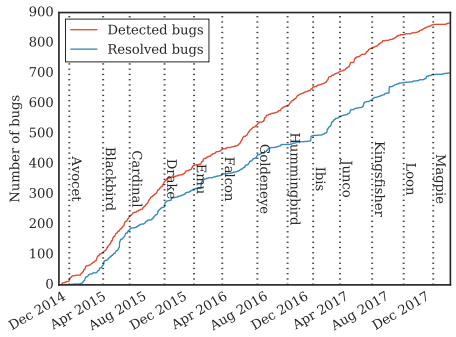
The SRGMs presented in the previous section, assume that the only changes in the code are due to the bug fixes, and hence, the bugs reports are separated based on the "affected release version" field. The number of the bugs reported for every release, grouped by the priority, are presented in Fig. 4.2c. Note that, due to the time overlap between the support periods some of the bug reports may affect more than one release. In the analysis of software maturity in Sec. 4.5, "minor" and "trivial" bugs (e.g., loading of the GUI too slow) were ignored, as they do not have an impact on the critical controller operations and often remain unresolved.

The most recent release at the time of the data set gathering, Magpie (ONOS v.1.12) did not have enough samples, i.e., bug reports, for the statistical analysis. Hence, the focus was on Kingsfisher (ONOS v.1.10), the most recent release whose support cycle had ended, and Loon (ONOS v.1.11), referred to as the two latest stable releases.

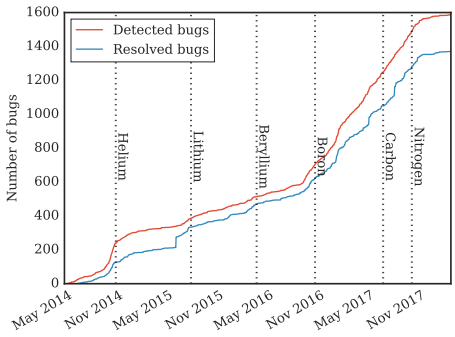
The distributions of the TTF and the TTR for Kingsfisher and Loon with the previous ONOS releases are compared in Fig. 4.2e. The median TTF was approximately 48 h, and was consistent for all three data sets. However, the median TTR showed higher variation, between 168 h to 180 h, or around a week. Both TTF and TTR show the characteristics of long tail distributions, which makes it difficult for the software management team to estimate, e.g., the effect of extended testing effort on the improvement of the software quality. The SRGMs presented in the previous section, add the time dimension to these distributions, enabling more accurate forecasting of dependability attributes, such as the expected number of bugs to be detected in a given time period.

4.4.2 ODL Dataset

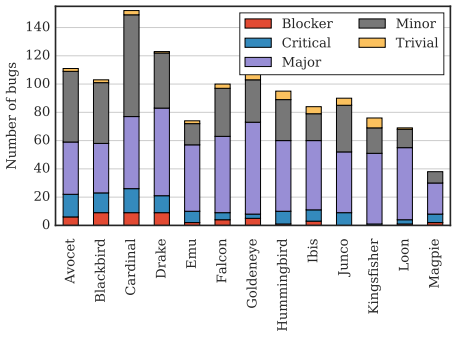
ODL has a much larger and complex code base, since its use case was initially much broader. ODL aim to provide the support for virtually all legacy networks, in contrast to ONOS which was primarily developed for service providers (see Sec. 2.2). The release management cycles of the two controllers are also different: while ONOS distributes the code in the regular three-month cycles, the lifecycle of ODL releases is irregular, between three and nine months, as illustrated in Fig.4.2b. Seven releases have been distributed from April 2013 to February 2018. The releases are named by elements in the



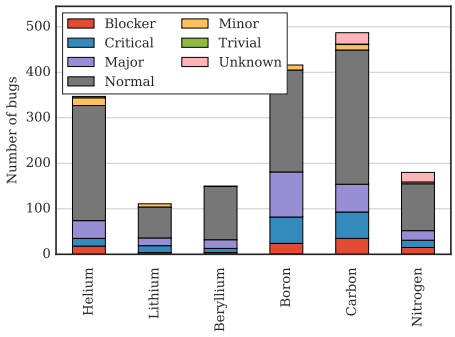
(a) Bug reports over time (ONOS)



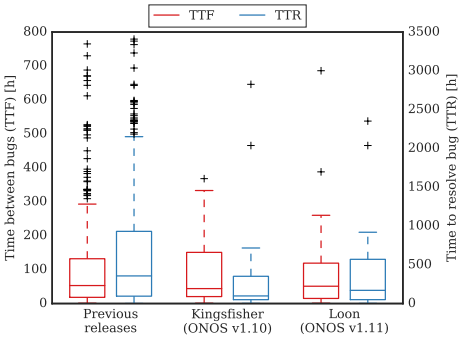
(b) Bug reports over time (ODL)



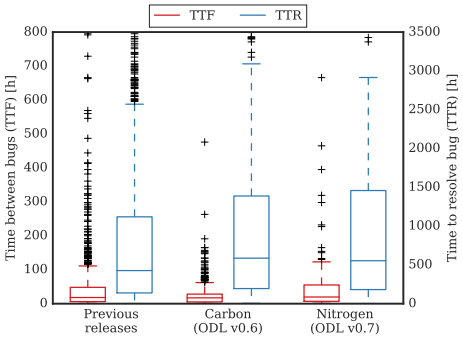
(c) Bug severity per release (ONOS)



(d) Bug severity per release (ODL)



(e) TTF and TTR distributions (ONOS)



(f) TTF and TTR distributions (ODL)

Figure 4.2: A first look at ONOS and ODL data sets: descriptive statistics of issues reported in the period between December 2014 and February 1, 2018.

periodic table. The bug reports for the first release, Hydrogen (distributed in February, 2014) are not included in the statistics, because only few bugs were reported at the beginning of ODL project.

The two controller platforms had a different approach to their issue tracking systems. While ONOS has been using Jira since its inception for the documentation and management of its bug repository, ODL relied at the very beginning on the internal mailing list and excel sheets, then used Bugzilla issue tracker in the first 6 releases, and migrated to Jira in October, 2017. Although both issue tracking systems offer the same reporting capabilities, ONOS bug reports provide higher level of detail and less ambiguity. An example is the classification schemes for bug severity. While ONOS has five well

defined categories, ODL has six, with majority of the bugs (68%) belonging to the default "normal" category. Some bug entries in ODL issue tracker are even left unclassified, as it can be seen in Fig. 4.2d.

The distributions of TTF and TTR in ODL releases are presented in Fig. 4.2f. The distribution of TTF is comparable to ONOS, while the distribution of TTR shows much larger variance.

Two particular releases, ONOS v.1.10 (Kingsfisher) and ODL v.0.6 (Carbon), are compared in Table 4.2. The releases were distributed approximately at the same time (June 5, 2017 and May 25, 2017, respectively) and sufficient time had elapsed for both controllers to reach the stable phase. The observed fault density per release, i.e the number of the bugs detected during the software lifecycle per lines of code reported for a particular release, of the two controllers is close to 0.1 $[\frac{\text{bugs}}{\text{kLOC}}]$, with ONOS having slightly lower fault density.

Table 4.2: Comparison of ONOS v.1.10 (Kingsfisher) vs. ODL v.0.6 (Carbon) releases

Controller	ONOS	ODL
Release	ONOS v.1.10	ODL v.0.6
Started	June 5, 2017	May 25, 2017
Lines of Code (LOC)	743,531	3,860,347
Reported bugs	76	493
Fault density $[\frac{\text{bugs}}{\text{kLOC}}]$	0.128	0.102

Note that the release fault density per release (Table 4.2) is different than fault density per software lifecycle (Table 2.1), since the latter accounts for the bugs reported during the entire software lifecycle, not just the particular release.

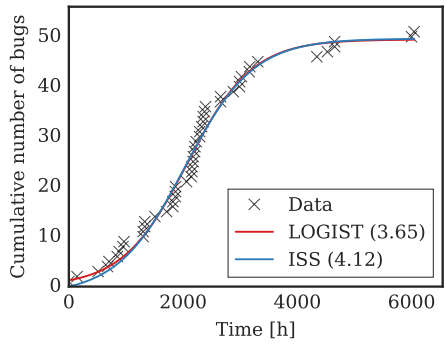
4.5 Best Model Selection

Once the data has been collected and preprocessed, the best fitting SRGM to describe the stochastic behaviour of the data set has to be found. The best SRGMs for the bug detection and bug resolution processes are discussed in Sec. 4.5.1 and Sec. 4.5.2, respectively.

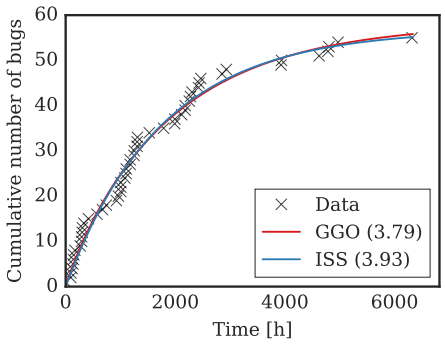
4.5.1 Bug Detection Process

The eight most widely used SRGMs for the bug detection process presented in the Table 4.1 are shortlisted as candidate models. The empirical data, i.e. the cumulative number of detected bugs, and the estimations of the two best fitting models for two stable releases of ONOS and ODL are presented in Fig. 4.3. The models are ranked based on the MSE, as it was the optimization criteria of the parameter fitting procedure (see Sec. 4.3.3), which is also indicated in the figure. Time-axis indicates the relative time since the beginning of the first reported bug for a release.

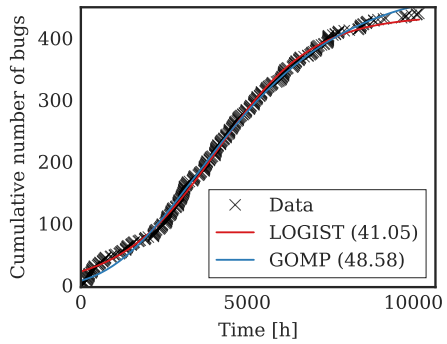
The analysis shows that all 3-parameter S-shaped models, Generalized Goel-Okumoto, Inflection S-Shaped, Gompertz and Logistic, fit the data well. Since the difference in MSE between these models is rather small, the estimated number of bugs for the two best fitting models is shown. The concave models, i.e. Musa-Logaritmic, Goel-Okumoto Exponential and Yamada Exponential, could not explain the data, except for the few releases (Avocet, Falcon, Loon and Beryllium) that experience more concave pattern.



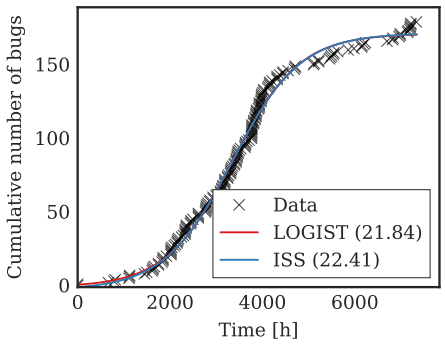
(a) ONOS v1.10 (Kingsfisher)



(b) ONOS v1.11 (Loon)



(c) ODL v0.6 (Carbon)



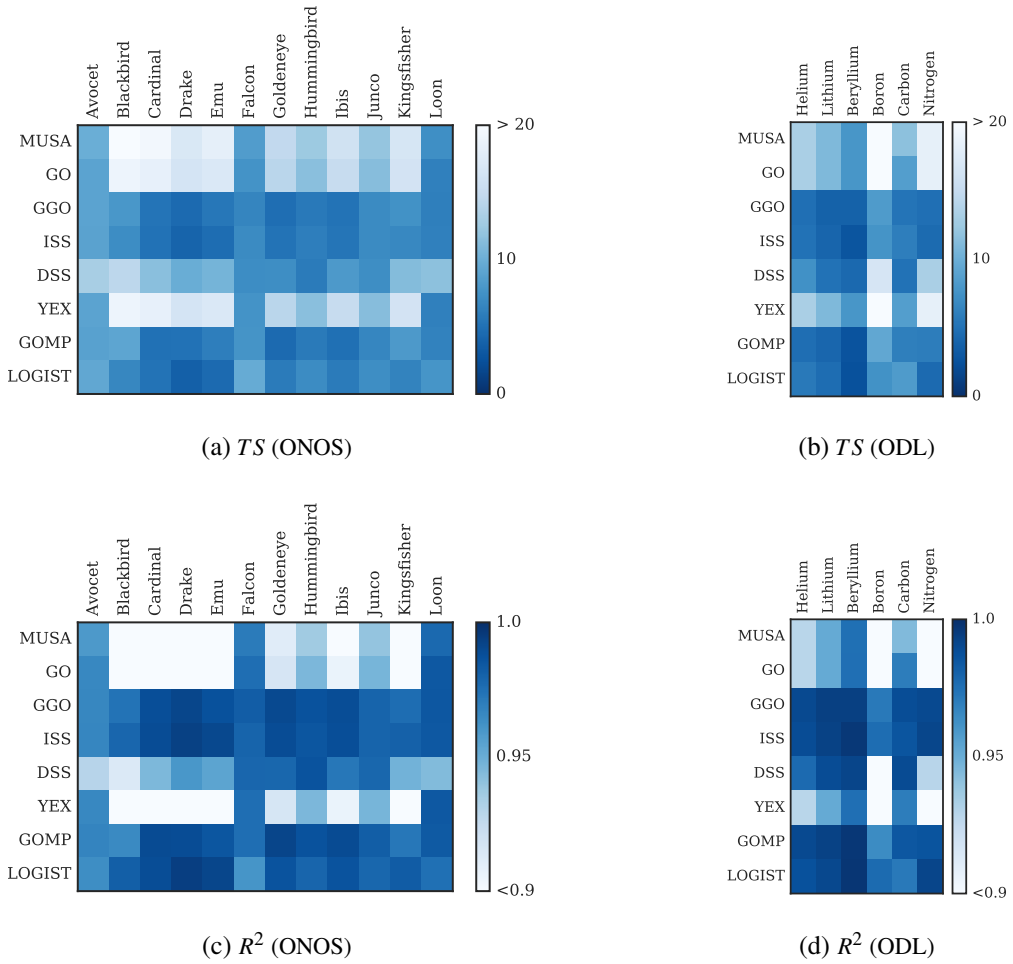
(d) ODL v0.7 (Nitrogen)

Figure 4.3: The best fitting models for bug detection process for stable ONOS and ODL releases.

The GoF metrics for all the models and the releases are compared in Fig. 4.4. All GoF indicators show consistent results: the best model to describe the number of detected faults across all releases are 3-parameter S-shaped models, showing very good scores in each metric. The best fitting models in the most of the cases are Logistic and Gompertz, followed by Generalized Goel-Okumoto. Inflection S-shaped model also shows very good GoF results, being the second best fit for most of the releases (for 12 out of 18 releases). Delay S-shaped shows slightly worse results, compared to the other S-shaped models. This effect is due to the fact that this model has only two parameters to tune, one less than the other S-shaped models.

4.5.2 Bug Resolution Process

An arbitrary combination of NHPP models can be used for fitting of the cumulative number of resolved bugs applying the Eq.(4.10). Here, the four combinations of S-shaped models are presented: Generalized Goel-Okumoto (GGO) and Inflection S-shaped (ISS). The models are abbreviated as a combination of the initials of detection and correction NHPP processes. For the sake of comparison we also include the reference model from [91] where both fault detection and resolution are modelled as Goel-Okumoto processes, which is the most widely used model due to the analytical tractability of the distributions for the combined process.



(a) TS (ONOS)

(b) TS (ODL)

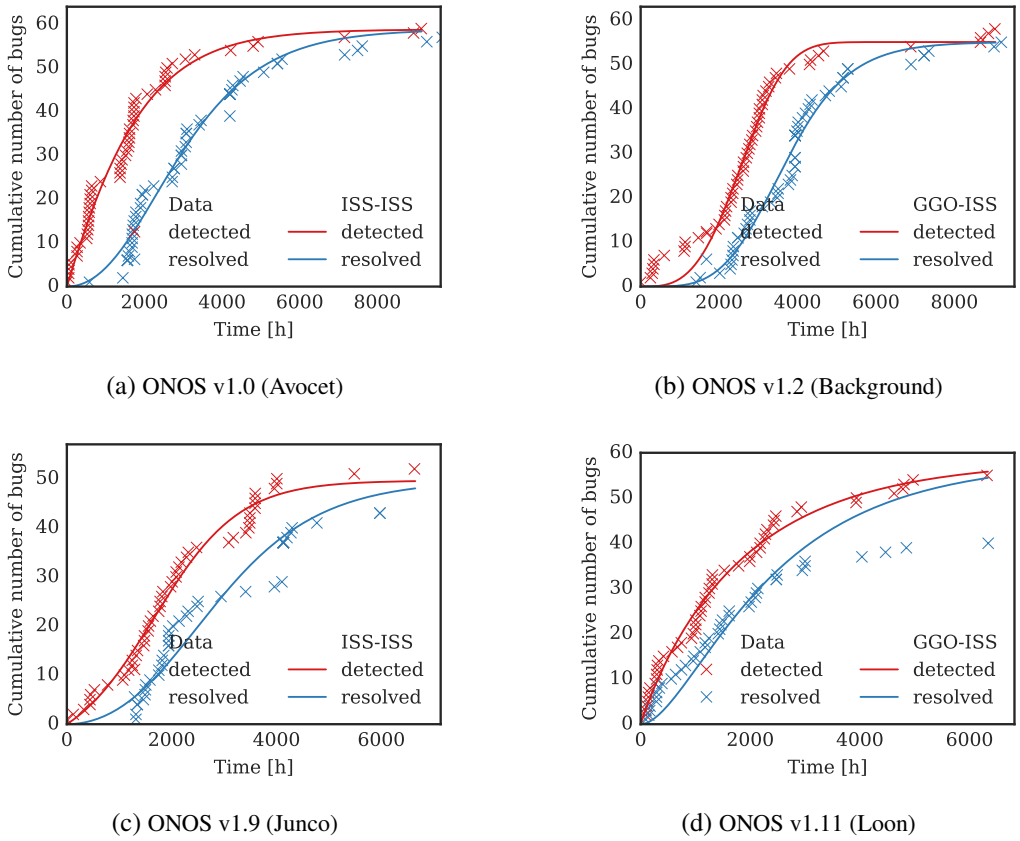
(c) R^2 (ONOS)

(d) R^2 (ODL)

Figure 4.4: GoF metrics: Theil's Statistics (TS) and Coefficient of Determination (R^2) for all ONOS and OpenDaylight releases.

The best fitting model for four representative releases, Avocet, Blackbird, Junco and Loon, are shown in Fig. 4.5. It can be seen that although the proposed models for the fault resolution process could describe the data for some of the releases, the actual data shows higher deviation from the fitted model, than in the previous case. In the first two cases (Fig. 4.5a and Fig. 4.5b) the models fit better the data. The best fitting models are ISS-ISS and GGO-ISS.

The two other releases, Junco and Loon, have experienced sudden trend changes around the official release date. In the case of Junco (Fig. 4.5c) two sudden increases can be detected: the fist one happens around its official release and the second one shortly before the distribution of the subsequent release. Similar behaviour can be observed in several other releases (Goldeneye, Hummingbird, Ibis). Such sudden trend changes due to external signals cannot be captured by a simple combination of NHPP models. The trend shifts due to the changes in a debugging effort shortly before the new upcoming release, can be modelled by introducing the time change points in the underlying NHPP models, as described in [76]. This approach requires the time change points to be provided either manually or defined as additional unknown parameters of the model. In the first approach, the generalizability of the model is poor, while in the second approach the estimation of the parameters in the small data



(a) ONOS v1.0 (Avocet)

(b) ONOS v1.2 (Background)

(c) ONOS v1.9 (Junco)

(d) ONOS v1.11 (Loon)

Figure 4.5: Comparison of the best fitting models for fault resolution process for four representative releases.

sets might be noisy (e.g., fitting the model with five or more parameters to dataset with less than 30 samples).

In the case of Loon (Fig. 4.5d), the trend after the official release is changed, indicating the change in the debugging strategy. Similar behaviour can be observed in Cardinal, Drake, Emu, and Falcon. It has to be noted that in open source software, such as ONOS, all the users are at the same time the testers, as anybody can report bugs in the public issue tracker. However, only a limited group of people will work on actually fixing the bugs. When this discrepancy between the "test" and "debug" team is too large, or when there is a sudden change in the size of debugging effort, the time scales have to be adjusted accordingly. The models, such as [147], can capture the changes in the test effort, but have the same problem of the accuracy of the parameter fitting on comparatively small data sets.

The same pattern can be observed also in the Fig. 4.6, where the MSE metrics of the five proposed models for all releases are compared. It can be observed that GGO-ISS and ISS-ISS outperformed the reference GO-GO model, for all the releases, where fitting was possible.

4.6 Software Maturity Assessment

This section presents the applications of SRGMs for the software maturity assessment, illustrating its practical value in three software management problems. The first problem addresses the optimal software release and software adoption times, based on the reliability and cost criteria, which is a

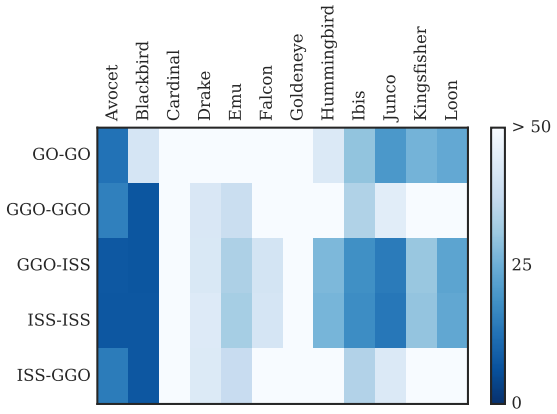


Figure 4.6: Comparison of MSE of SRGMs for the bug resolution process.

typical use case of SRGM found in the literature [78, 90, 95, 96, 99, 194, 195]. Next, two novel use cases are presented, relevant for the SDN community, showing how analysis of SRGM parameters can be used for (i) an early estimation of software reliability, and (ii) as criteria to discriminate between alternative controller platforms, e.g., ONOS and ODL, when reliability has the highest priority.

4.6.1 Optimal Software Release and Software Adoption Time

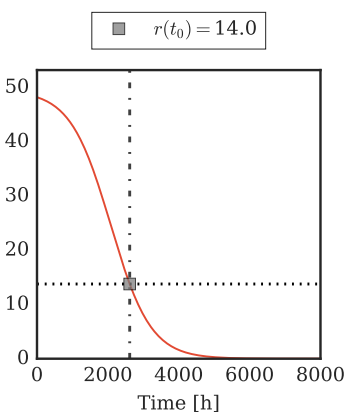
SDN controllers comprise all the functionalities of the network operating system, and require constant updates to keep up with the evolution of the user requirements [63]. Rapidly changing complex software systems are prone to bugs, which can be detected and removed only through extensive testing. Hence, it is of the utmost importance to provide tools to quantify the maturity of the software and determine when it is ready to be released and integrated into production networks. This section discusses how SRGMs can be used to to estimate the quality of controller software, and determine the optimal software release and adoption time, based on the software reliability and the cost criteria.

4.6.1.1 Software Reliability Criteria

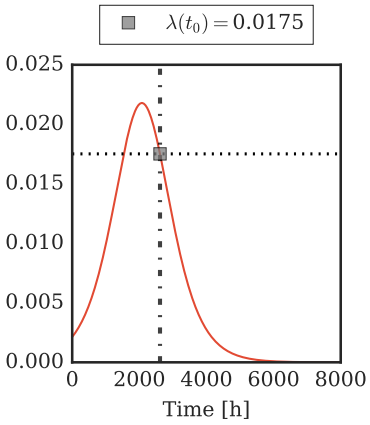
Software reliability, defined in the literature as the probability of failure-free software operation for a specified period of time in a specified environment, is an important indicator of software quality. Once the best model to describe the fault report data is selected and the parameters are estimated, it can be used to predict software reliability metrics: expected failure rate, residual bug content and conditional software reliability, as defined in Sec. 4.3.1 by Eq.(4.6), Eq.(4.5) and Eq.(4.7) respectively.

The software reliability metrics for the Kingsfisher release are presented in Fig. 4.7. Kingsfisher is the most recent ONOS release whose support cycle has ended at the moment of data set collection. The best fitting model is Logistic, as illustrated in Fig. 4.3a. The official release date t_0 is indicated with the vertical line in the figure, and the time is expressed as the relative time since the start of the testing. Note that only severe bugs (bugs with major, critical and blocker priority) are considered.

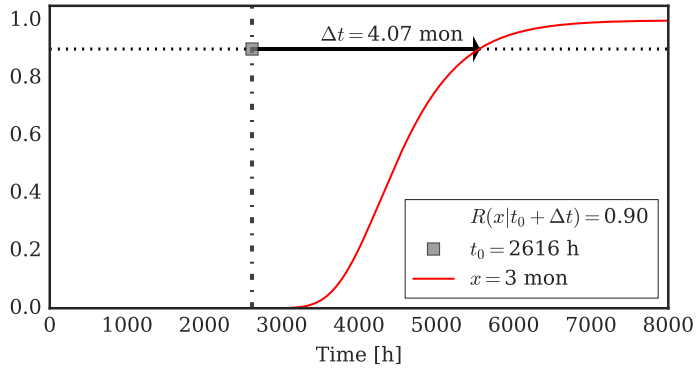
Residual bug content represents the number of undetected faults remaining in the software. It can be seen in Fig. 4.7a that the residual bug content was relatively high, as 14 severe bugs were still



(a) Residual bug content $r(t)$



(b) Failure intensity $\lambda(t)$



(c) Software reliability $R(x|t)$

Figure 4.7: An example of the optimal software adoption and release time based on the reliability criteria. Vertical lines indicate the date of the official Kingsfisher release time (t_0).

remaining in the software on the day of its official release. Three months after the official release, the number of remaining bugs is expected to drop to zero.

Instantaneous failure intensity, or alternately expected time until the next software failure, can be derived from the parameters of the mean value function. The expected failure intensity, illustrated in Fig 4.7b, on the day of Kingsfisher's release was at the level of $0.0175 \frac{\text{bug}}{\text{h}}$, or equivalent to approximately 2.38 days between detection of successive severe bugs. The fault intensity is highly relevant for the software developers, as it can indicate when the software is ready for the release. This metric could help the developers estimating the efficiency of investing the additional testing effort.

Conditional software reliability represents the probability of not encountering any software failure in the time interval $[t, t + x)$. The time interval of interest starts with the software adoption time t, with a duration of x time units, specified by the user's desired warranty period. It can be observed that in order to achieve reliability of $R(x|t) = 0.90$, during maintenance interval of $x = 3$ months, the user should defer the software adoption more than $\Delta t \geq 4$ months after its official release t_0, as illustrated in Fig. 4.7c. Note that the recommended adoption deferral period of 4 months is larger than the 3-month gap between two consecutive ONOS releases. Nevertheless, it is a common practice in

telco and enterprise domains not to use the most recent, but the lagged version, due to the stability issue. Hence, ONOS provides the support for the latest two releases, implying the support window of 6 months after the official release date.

4.6.1.2 Software Cost Criteria

Software management needs to balance the effort spent on the testing in the pre-release phase, and effort spent on the bug removal of the software in the operational phase. Open source SDN controllers, such as ONOS and ODL, offer no guarantees on either performance or reliability. However, many commercial solutions provided by network vendors, such as Ericsson and Huawei, are built on top of these controllers.

The software cost model, defined by Eq.(4.8), generalizes most of the cost models proposed in literature. The testing cost $c_t(t)$ function accounts for the cost of the software testing team, the cost of the bug removal, the setup and the maintenance of the testing environment, the code documentation, etc. The cost during the warranty period $c_w(t)$ includes the penalty paid for every severe outage encountered during the normal operation, the cost of network service interruption, the cost of the bug removal and the support team and sometimes also a discounted value of money for the long support periods. These cost factors must be determined per use case basis. The cost functions $c_t(t) = C_t$ and $c_w(t) = C_w$ are assumed to be constant, i.e., independent of the bug complexity, which is common assumption in the literature [136, 194]. The constant factors represent the average cost of bug removal in test (C_t) and operational (C_w) phases. The software cost function from Eq.(4.8) then can be expressed as:

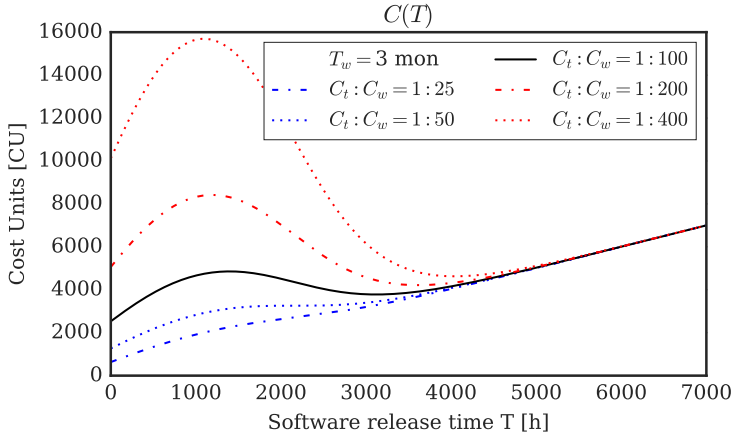
$$C(T) = C_t T + C_w[m(T + T_w) - m(T)] \tag{4.16}$$

where $m(t)$ is a mean value function of the best fitting model, discussed in the previous section. The optimal software release time T is obtained by finding the minimum of expected cost function. For the simpler models, e.g., Goel-Okumoto, the optimal solution, i.e. the minimum of the cost function $\frac{dC(T)}{dT} = 0$, can be found analytically, while for other models the minimum has to be found numerically.

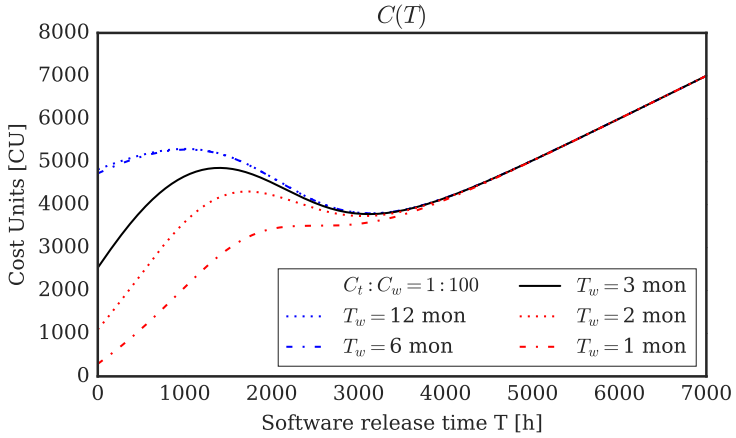
The relative cost (in unnamed cost units CU) of C_t and C_w is assumed to be $1[\frac{\text{CU}}{\text{h}}] : 100[\frac{\text{CU}}{\text{bug}}]$, while the warranty period of T_w is assumed to be 3 months in the baseline scenario. The impact of different $C_t : C_w$ and T_w on the software cost is illustrated in Fig. 4.8. In some scenarios the cost function has no clear minimum. In the cases when the total cost of bug removal in the oprational phase is expected to be low, either due to low removal cost C_w (Fig. 4.8a) or the very short warranty period T_w (Fig. 4.8b), the optimal software release policy is to distribute the software immediately. In the baseline scenario, a clear minimum for the software release time T can be observed, which is approximately 40 days after the official software release date ($t_0 = 2616h$), highlighting the benefits of the extended testing period.

4.6.2 Early Prediction of Software Reliability

In order to estimate the SRGM parameters, a large number of samples, i.e. bug reports, has to be provided. In case of ONOS, the standard parameter fitting techniques cannot accurately predict the model parameters before 90% of all bug reports are available, which happens approximately after six months of testing when it is already too late for software developers (as the software is already released)



(a) Impact of the relative cost of code testing (C_t) and bug removal (C_w) during release warranty period



(b) Impact of the duration of warranty period (T_w)

Figure 4.8: An example of optimal software adoption and release time based on the cost criteria, illustrated in the example of Kingsfisher release.

and the SDN operators (since new release is already available). Estimating the SRGM parameters when only few data samples are available, is especially difficult for S-shaped models, since they change the concavity around three months after the start of the integration testing (Fig. 4.3). However, it can be observed that the SRGM parameters show very small variation across the releases, thanks to the incremental development strategy of ONOS, as it can be seen in the case of the Gompertz model in Fig. 4.9. This knowledge can be used to guide the parameter fitting procedure, and regularize the model which improves the prediction accuracy in the early phase. The regularization of the model is implemented by restricting the parameter search space, as described in Sec. 4.3.3.

The trend observed in Fig. 4.9 shows several interesting points how the regularization of the search space can be done. The scale parameter a and the shape parameter b show comparatively small variations between the consecutive releases. The parameter a varies between 54 and 85; parameter b is in the range (0.99879, 0.99935). The parameter k for the releases with S-shape bug detection trend is

in the range (0,0.02). The releases with concave trend (Avocet, Falcon and Loon) show higher values of parameter k, in the range of (0.5,0.85).

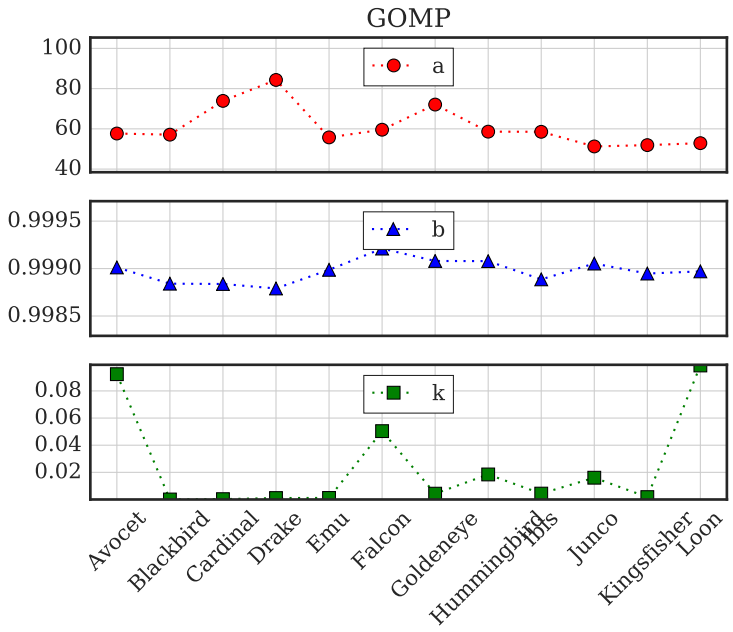


Figure 4.9: Estimated parameters of Gompertz model for bug detection process for all ONOS releases.

Several parameter regularization strategies have been explored. The first proposed strategy is based on the extreme values, where the search space of every parameter ξ is bounded to $[0.9\,\xi_{min},\ 1.1\,\xi_{max}]$, which represents the range of previously observed parameters extended by 10%. The second strategy based on the mean m_ξ and the variance σ^2_ξ, where the parameter search space is bounded to $m_\xi \pm 2\sigma_\xi$. The third proposed strategy is based on the parameter trend, considering an exponentially weighted moving average, which is defined as:

$$m^i_\xi \leftarrow \omega\,\xi^i + (1-\omega)\,m^{i-1}_\xi \tag{4.17}$$

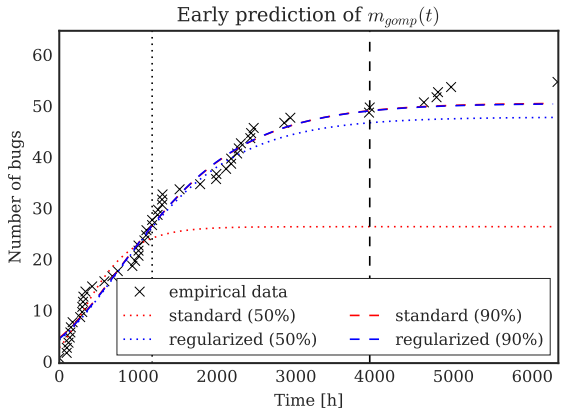
where the average value of the parameter ξ after i releases m^i_ξ is computed as a weighted sum of the estimated parameter for the i-th release ξ^i and the previous average value m^{i-1}_ξ. Here, a value of $\omega = 0.5$, is assumed, and the parameter search space is bounded to $m^i_\xi \pm 2\sigma_\xi$. Note that in cases where the lower bound is negative, the values are capped to zero, due to the nature of the model. The parameter search space bounds with different preparation strategies are compared in Table 4.3.

Table 4.3: Gompertz model regularization with parameter prediction strategies, based on: i) extreme parameter values, ii) mean and variance and iii) moving average.

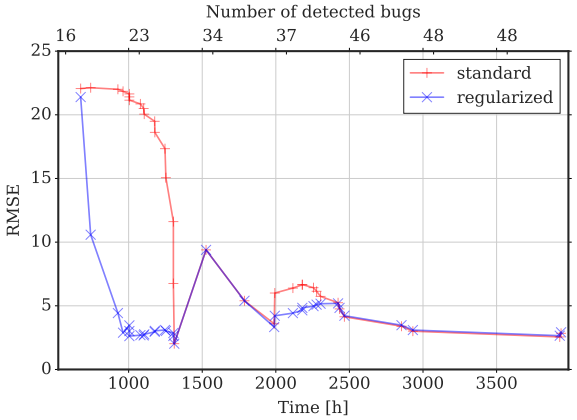
ξ	$[0.9\,\xi_{min},\ 1.1\,\xi_{max}]$	$[m_\xi \pm 2\sigma_\xi]$	$[m^i_\xi \pm 2\sigma_\xi]$
a	[50.74, 85.15]	[41.62, 80.65]	[34.21, 73.64]
b	[0.9888, 1.0092]	[0.9987, 0.9992]	[0.9982, 0.9987]
k	[8.2 e-7, 0.0933]	[0.0, 0.0936]	[0.0, 0.0629]

All prediction strategies narrow down the parameter search space: while the first strategy covers extreme values, the range for the other two are more narrow. Overall, all prediction strategies showed

improvement over the standard fitting techniques, demonstrating the positive impact of the prior knowledge on the parameter fitting accuracy. The prediction strategy based on the trend shows the unstable performance when parameter experienced the sudden trend changes, as in case of parameter k for Loon release, illustrated in Fig. 4.9. It might be possible to use a different regularization strategy for each parameter. As a part of the future work such hybrid strategies will be considered, which might have better performance after more software releases are available and behavioural patterns of each parameter can be estimated more precisely.



(a) Early prediction of mean value function $m_{gomp}(t)$, when limited number of samples are available.



(b) Evolution of Root Mean Square Error (RMSE) with the number of training samples.

Figure 4.10: Early prediction of software reliability, when only few samples, i.e. bug reports, are available for the fitting of SRGM parameters. Benefits of regularization can be seen in the evolution of mean value function and RMSE, as illustrated for Loon release and Gompertz model.

The benefits of regularization on the early prediction of software reliability, can be quantified by observing the estimated mean value function $m_{gomp}(t)$ and the evolution of Root Mean Square Error (RMSE), when the limited number of the samples, i.e. bug reports are fed to the parameter fitting function. The results for the prediction strategy based on the observed mean and variance are shown in Fig. 4.10. The impact of the error in parameter estimation is illustrated in Fig. 4.10a. The error of

the estimation with 50% of the available samples with standard fitting techniques is much larger due to the local variations of early samples. It can be observed in Fig. 4.10b that the regularized model is able to estimate the parameters with higher accuracy much earlier, with 30% fewer samples. While the standard fitting technique requires 32 samples for RMSE to drop below 3, the regularized technique only needs 21 samples.

The results in the figure are presented for the Gompertz model, which has the best performance across all releases, being the best fit for five releases, and showing very good results for the other seven. Moreover, the parameters of Gompertz model have shown the smallest coefficient of variation (variance/mean). However, the general conclusions hold for the other three 3-parameter S-shaped models, as well. While studying the impact of the model selection, it is observed that, in general, the regularization improves the predictive capabilities of SRGM in the early phase of the software lifecycle for all 3-parameter S-shaped models, but the magnitude of the improvement depends on the data set. For Junco release, none of the combinations of the models and prediction strategies show significant improvements with 50% of the samples. This is probably due to the timing of the burstiness of bug reports at the beginning of testing (see Fig. 4.3). Further improvements could be achieved with smoothing techniques and grouping of the data, e.g, by reducing the time resolution of the bug reports from hours to days or weeks. The limitations of SRGM are further discussed in Sec. 4.7.

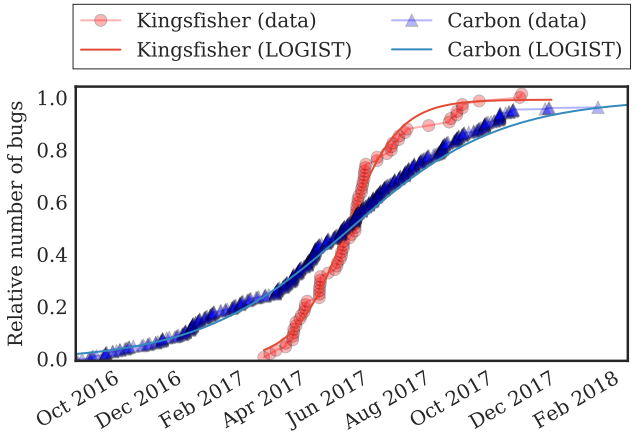
4.6.3 Software Maturity Metrics: Comparison of ONOS and ODL

As SDN is gaining the popularity a multitude of commercial and open source SDN controllers have been developed. While most of the early open source solutions have remained in the research community at prototype level, two projects have reached production grade readiness: ONOS and ODL. In this section, the controller selection problem is addressed, relevant for a network operator choosing the optimal commercial SDN controller platform for its network, or alternatively, an open source platform as a code base to build his customized controller upon, when code maturity is the major concern. Although the difference in the support of some of the advanced networking features is still present (e.g., the ODL support for the wireless networking), the two controller platforms are converging and it is not clear for network operators, which solution to choose based only on the supported functionality. For instance, the commercial SDN controller platform by Ericsson is based on ODL, while Huawei Agile controller solution is based on ONOS, and AT&T deploys both platforms in its production networks.

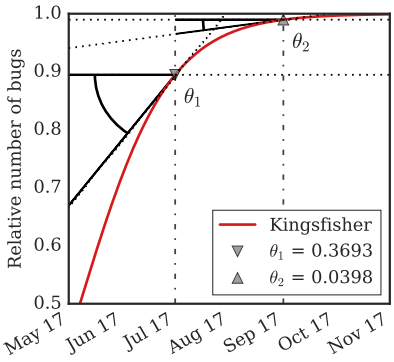
The release fault density presented in Table 4.2 is a static measure of the code quality, which can be reliably computed only after the software lifecycle is over and the support has ended. Several methods have been proposed for an early estimation of fault density, based on the complexity, used programming language and other software features, which might not always be available to the public. On the other hand, the SRGM framework treats the software component as a black box, and provides the estimation of the software reliability without requiring the information about the code internals. The challenge of direct comparison based on the empirical data between the two releases is illustrated in Fig. 4.11a[3]. It can be observed that the direct comparison of the empirical data is not straightforward (assuming that the bug detection is a realization of the stochastic process), and that on June 1, 2017, both controllers

[3]In software maturity analysis bugs of all priorities reported during entire lifecycle are included in the analysis, as illustrated in Fig. 4.11 and 4.12, while in Fig. 4.3 only severe bugs (ONOS) and bugs reported after the start of the integration testing (ODL) are included.

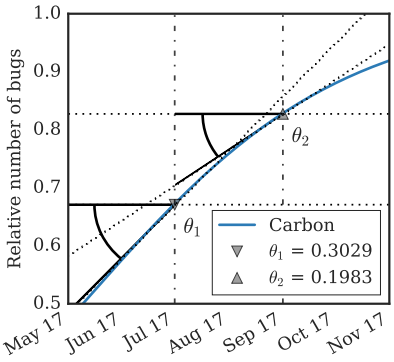
had detected around half of the number of bugs. Instead, it is much more precise to compare the fitted curves of the two controllers in order to project the expected number of bugs in the future.



(a) Emprical and fitted data for the two controllers.



(b) Software maturity of Kingsfisher: one (θ_1) and three months (θ_2) after the software release.



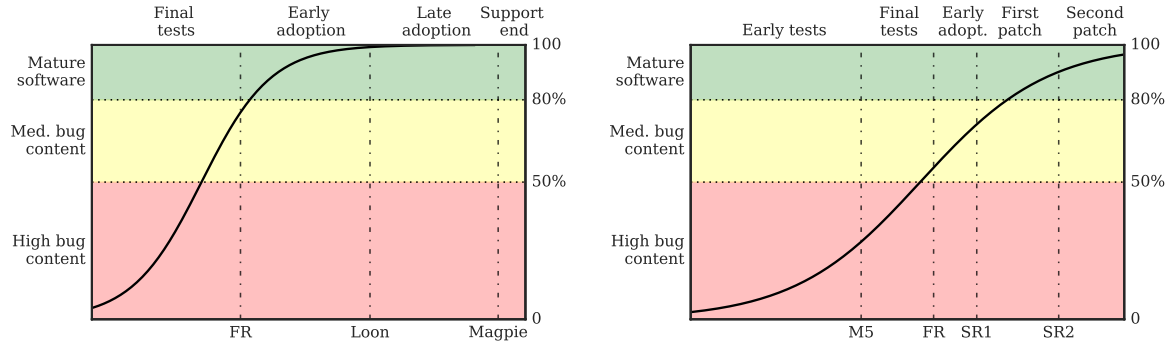
(c) Software maturity of Carbon: one (θ_1) and three months (θ_2) after the software release.

Figure 4.11: Software maturity evolution over time for Kingsfisher (ONOS v1.10) and Carbon (ODL v0.6).

In order to compare the reliability and code stability of the two SDN controllers, this work proposes the *software maturity metric*. The software maturity metric is derived from the respective SRGM as a scaled gradient of the cumulative number of bugs, i.e., $\lambda(t)/m_{max}$, which provides a measure on how far is the software from the stable region (i.e. how close to horizontal line) at any given moment. The practical value of the proposed software maturity metric is illustrated in Fig. 4.11b and Fig. 4.11c, where the software maturity after one (θ_1) and three months (θ_2) after the official software release is indicated. The units are expressed as the percentage of detected bugs per day, where zero indicates the stable software. It can be observed that the maturity of Kingsfisher improves much faster $\theta_1 = 0.3693[\frac{\%}{\text{day}}] \rightarrow \theta_2 = 0.0398[\frac{\%}{\text{day}}]$, compared to the Carbon $\theta_1 = 0.3029[\frac{\%}{\text{day}}] \rightarrow \theta_2 = 0.1983[\frac{\%}{\text{day}}]$), thanks to the shorter release lifecycle of ONOS.

The software maturity metric can be further used to profile the behaviour of the controller, and quantify the improvement of the software quality over different software lifecycle phases, as illustrated in Fig. 4.12. The comparison of the maturity evolution over time across different releases can be used

to track the progress of the software development process and the efficiency of the testing effort on the improvement of software quality.



(a) Kingsfisher lifecycle: the date of the formal release (FR), subsequent release (Loon) and the end of support (after Magpie release) are indicated in the figure.

(b) Carbon lifecycle: the start of integration testing (Milestone M5), the date of the formal release (FR), and the release of software patches (SR1 and SR2) are indicated in the figure.

Figure 4.12: Software maturity in different phases of the controller lifecycle.

The challenging part of this process is the early estimation of m_{max}, which have to be estimated before the software lifecycle is over. The approach for an early estimation of SRGM presented in Sec. 4.6.2 can be used to evaluate m_{max}. Note that in this particular case, at least 50% of the bug reports were available before the official software release dates for both controllers, in which case our approach for an early prediction can estimate the SRGM parameters with the reasonable accuracy.

4.7 Concluding Remarks

4.7.1 Summary

This chapter presents a framework to assess and to predict the maturity of SDN controllers, extending the classical SRGMs. Using real data on software failures of the SDN controllers ONOS and ODL, SRGMs describe the stochastic behaviour of bug manifestation and correction processes, which makes it possible to analyze the reliability of controllers' software releases. The investigated software reliability metrics derived with SRGM can be used to guide software developers and network operators to help making important operational decisions: e.g., deciding on when a software controller is mature enough to be released and deployed. The model regularization techniques for the early prediction of software reliability based on transfer learning, i.e., the observed trend of the model parameters of previous software releases have been proposed. Moreover, novel software maturity metric has been defined, which can be used as a selection criteria for controller software candidates.

4.7.2 Discussion

Threats to validity. The presented framework imposes few limitations on the prospective use cases. The first limitation comes from the fault reports, as the results are only as good as the accuracy of the data sets. SRGMs require the complete and uncensored fault reports, in order to accurately estimate the parameters in the model. Since neither the accuracy nor the completeness of the reported

data in the public issue trackers can be guaranteed, the focus is shifted from the quantitative analysis to the general approach to quantify and forecast the software reliability. The second limitation stems from inherent assumptions of SRGMs. The models assume independent times between the consecutive fault reports, as well as that every undetected fault contributes the same to the fault manifestation rate, which is not entirely met in practice. Although there is no guarantee that any network software can be modelled as mixture of simple SRGMs, previous studies have shown that described models can be successfully applied to many large open source software products, such as Apache Web Server, Mozilla Firefox, and Eclipse IDE (Sec. 4.2). In order to benefit from early prediction method, there are several important dependencies: first, a relatively large number of regular releases has to be available; second, the behaviour of the releases has to be similar enough, which has been the case only for ONOS so far. However, as the number of releases increases for other SDN controllers, the regularity of the release distributions is expected to stabilize (observed in other open source projects, e.g., Linux OS). Consequently, the early prediction approach might find more valuable applications for further SDN controllers in near future.

Generalization of the results. The main practical value of the proposed approach lies in the applicability of the framework for the assessment of the software maturity of network control software. As demonstrated for two open source SDN controllers, ONOS and ODL, the analysis can be applied to other open source products or even the commercial products of major vendors (e.g., Huawei or Cisco) which build up on ONOS and ODL code base. Moreover, bug reporting systems of the controllers' collaborative software projects provide a valuable source of data that can be used by our framework. For instance, developers can report bugs either directly via code version control systems like Git or in separate issue trackers, e.g, the open source Bugzilla or the commercial Jira tracker. Furthermore, the described workflow can be easily integrated into existing AGILE software development techniques, such as SCRUM; hence, enabling the developers even receive continuous feedback on the quality of their code and efficiency during development process.

Future work. Although the SRGMs have shown to be a powerful tool for forecasting of the software defects in network control software, they suffer from two main drawbacks. First, the early prediction is often inaccurate and large number of samples need to be collected before a reliable model parameter estimation can be made; second, the categorical data used for the identification of critical defects is often subjective. Further enhancements can be achieved by improving the predictive power of reliability growth models with Artificial Neural Networkss (ANNs), and automating the classification and analysis of the large corpus of software bugs with Natural Language Processing (NLP).

Improving predictive power of reliability growth models with ANN. Estimating SRGM parameters in early operational phase, may lead to inaccurate estimations when standard statistical inference techniques are used. ANN architectures, such as Recurrent Neural Networks (RNN) can be used to improve predictive accuracy of SRGM in two ways. First, RNNs are better in separating long term trends from short term variations, which was a challenge for traditional SRGM models. Second, RNNs are expected to benefit more from the transfer learning, and can be trained on the similar projects with larger data sets.

Improving accuracy of software defect prediction and defect removal with NLP. The accuracy of fault forecasting with SRGM depends on the accuracy of the empirical dataset, i.e., the reports on the history of previous failures. The historical failure reports are often subjective opinions and hence,

often contain inaccurate information [118]. NLP can be used to reduce the noisiness of the subjective categories, such as defect severity, by automatic classification the bug reports based on the free-text input, and other categorical attributes. Another application of NLP is automatic knowledge extraction from large corpus of free-text descriptions in bug reports and error logs, which can guide the developers towards faster defect localization and removal [150].

Chapter 5

Dependability Assessment Framework for Distributed SDN Implementations (DASON)

5.1 Introduction

5.1.1 Motivation, Problem Scope and Research Challenges

In SDN, network programmability is enabled through logically centralized control plane. Production networks deploy multiple controllers to ensure scalability, high availability and high performance. In distributed control plane architectures, the benefits of logically centralized network control are preserved by means of distributed protocols, such as Gossip and Raft [139]. However, correct and stable implementation of distributed network control plane is not trivial, as confirmed by Google's report on critical network outages [63], which showed that control plane issues prevail in their B4 WAN[1]. Their analysis showed that under *control plane software failures*, *maintaining globally consistent network state* is difficult, and the *cascade of control-plane element failures* is a common culprit of critical customer impacting failures.

Defects in the implementation of distributed network control plane are particularly critical, given that such architecture is supposed to provide high availability, a key enabler for mission critical applications. Nevertheless, the state of the art literature is still missing a comprehensive study on the prevalence of failures in imperfect distributed SDN orchestration platforms. The goal of this study is to close this gap by systematically studying the nature of such failures and providing high fidelity models that can reproduce the stochastic behaviour of real-life distributed SDN platforms. Such models are needed in order to identify dependability bottlenecks, and reliably assess whether SDN solutions are ready to be deployed in a particular use-case scenario, such as industrial networks [7]. The controllers in this study are ONOS [28] and ODL [119], two of the largest distributed production-grade open source SDN orchestration platforms, whose code internals and bug repository are publicly available, allowing us to perform an in-depth dependability assessment.

[1] B4 [86] Google's internal WAN, carrying the traffic between data center clusters, is arguably the biggest live SDN network, both in geographical scale and the volume of traffic it serves.

5.1.2 Methodology: Data-driven Stochastic Reward Nets (SRN)

Dependability assessment framework for distributed SDN implementations, named DASON, is proposed in this chapter. The goal of DASON is a holistic assessment of system dependability, both in the control plane and service plane level. DASON is *data-driven* dependability assessment framework based on Stochastic Reward Nets (SRN), as illustrated in Fig. 5.1. The framework implements a general *analyse-model-evaluate* meta-workflow for dependability assessment, and it is applied to the use-cases of open-source distributed SDN orchestration platforms.

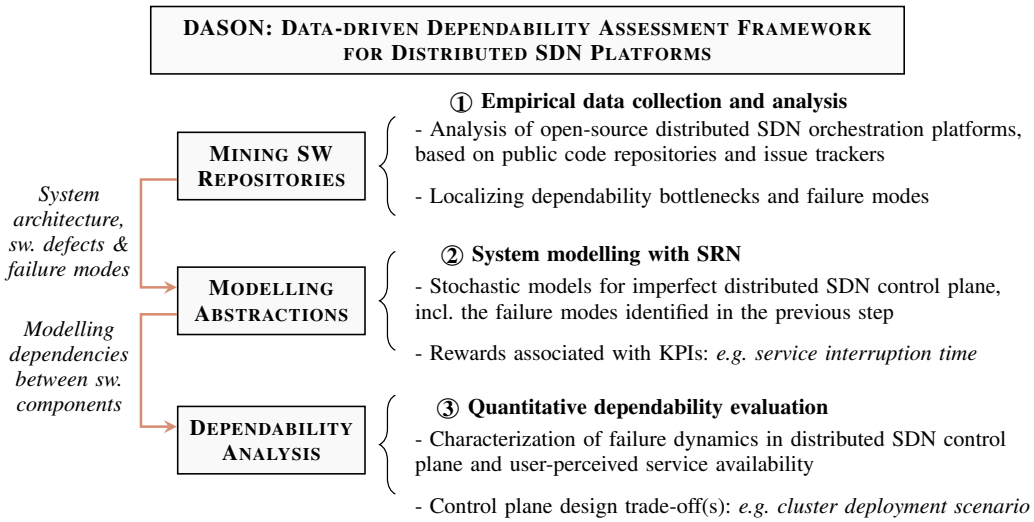


Figure 5.1: An overview of DASON: Data-driven dependability assurance framework based on Stochastic Reward Nets (SRN).

1. Mining software repositories. In the analysis step, the system architecture and failure modes can be extracted by mining software repositories of two distributed SDN platforms, ONOS and ODL, whose code repositories and issue trackers are open to the public, facilitating the in-depth analysis of system vulnerabilities.

2. Modelling abstractions. The modelling abstractions are defined in the formalism of SRN [75]. SRNs can be directly mapped to Markov chains, and are widely used in modelling complex systems consisting of large number of dependent components. The modelling abstractions for single controller nodes, their interaction using distributed system protocols, as well as the services that run in such architectures are provided. Dependability KPIs of interest, e.g., downtime distribution and outage frequency, are computed by assigning reward rates at the SRN. Model parameters are based on real-life controllers and systems of a similar function and complexity.

3. Quantitative dependability evaluation. Once the stochastic models are parametrized, they can be used to characterize the failure dynamics in a distributed SDN control plane, as well as their impact on user-perceived service availability. Different control plane designs, e.g., cluster size and deployment scenarios, can be compared easily, by modifying the parameters of the stochastic models.

5.1.3 Key Contributions

The contributions of the work presented in this chapter can be summarized as following:

i. A comprehensive analysis of real-life distributed SDN implementations is conducted, with the goal of localization of software defects and prevalent failure modes.

ii. The modelling abstractions for an imperfect distributed control plane, and its interaction with the service plane are proposed, as well as the reference models of a stand-alone controller deployments and preventive maintenance policies.

iii. The parametrized models are used to characterize the failure dynamics in realistic scenarios, including not only pure control plane dependability metrics, but also user-perceived service availability.

The results presented in this chapter have been published in peer-reviewed journal [6], and conferences [14, 16]. The first SRN model of an SDN controller was proposed in [14]. However, this first model did not address the interactions between distributed controller replicas, a leading cause of the recent controller failures, as shown in [16]. Hence, in [6] more accurate modelling abstractions for imperfect distributed SDN implementations are provided. Moreover, the previous study [16] is extended with taxonomy and systematic analysis of the distributed control plane defects.

The remainder of this chapter is organized as following. Sec. 5.2 provides an overview of the limitations of related work on distributed SDN controller frameworks, and relevant model-based studies. Sec. 5.3 presents an overview of distributed SDN control planes, while Sec. 5.4 discusses defects of real life distributed SDN implementations. In Sec. 5.5 modelling abstractions based on SRN for imperfect distributed SDN plane are presented, and are used for the quantitative analysis of control plane and service availability in Sec. 5.6. Sec. 5.7 concludes the chapter with a summary and discussion of the results.

5.2 Related Work

5.2.1 High-availability in Distributed SDN Implementations

This section presents an overview of the studies on distributed SDN platforms, identified issues in distributed control plane implementations, measurements-based benchmarks and evaluations of such implementations, as well as the large empirical studies that motivated the work presented in this chapter.

Distributed SDN platforms. A good overview of distributed SDN control platforms is presented in [26]. The survey compared different architectural designs and their approaches to address scalability and high-availability issues. However, most of the considered controllers have not made it into production environments, such as Onyx, HypeFlow, DISCO and Kandoo [70, 97, 148, 176], or are closed proprietary solutions, such as Google's B4 [86] and Espresso [197]. Hence, the focus of this chapter is on ONOS [28] and ODL [119], two production-grade open source controllers, which form the code-basis for many other commercial vendor products. The *software maturity* of these two distributed platforms, in terms of the reliability growth, has been compared in [5], as presented in the previous

chapter. The further mining of ODL software repositories [16] identified the clustering module, i.e., distributed control plane implementation, as the most vulnerable in terms of the number of software defects, but it did not investigate further the nature of such issues.

The issues in distributed SDN implementations. The issues in distributed SDN implementations have been addressed by several studies. *Stability under high load* of distributed control plane implementations with ONOS was analyzed in [68]. The authors have shown that consensus protocols, such as Raft, misbehave in overload conditions, due to increases in the delay of heartbeat messages and time-threshold based failure detectors. Such behaviour triggers frequent leader re-elections, leading to a crash of the entire control plane. The same effect of performance degradation under high load causing a *node flapping, repeated leader elections, and a cascade of control plane failures* was also observed in [44], which noted that the problem was already reported in the bug repository. Sakic et al. proposed ODL control plane enhancements, such as adaptive consistency [160, 161] addressing the issue of chattyness of consensus protocols, and Byzantine Fault Tolerance (BFT) protocols [159] addressing the *security and reliability issues of misbehaving controllers*. The mining of ONOS and ODL bug repositories, discussed in Sec. 5.4, exposes many more issues of practical distributed control plane implementations.

Measurement-based studies on distributed implementations. Two informal measurement standards on SDN control plane benchmarking, by IETF [29] and ONF [138], specify cluster performance and stability tests. The performance of ODL clustering, in terms of synchronization overhead, failure detection and failover time, was analyzed in [170], while ONOS inter-controller traffic in different scenarios was measured and modelled in [125]. An ONOS report on SDN control plane performance [140] discusses distributed design solutions considered by developers, as well as the final implementation, and demonstrates the improvements compared to an older release. These standard performance and cluster stability tests have already been incorporated in the ODL and ONOS test suites.

Despite the extensive testing many of the bugs go unnoticed and manifest only in the production environment. One of the reasons why many bugs escape the testing phase is *non-determinism*, such as *racing and concurrency issues*, which makes them extremely hard to reproduce, since triggering them requires precise timing between input events and internal procedures [107]. In [69, 123] the authors showed a huge number of concurrency violations in SDN controller applications. In the follow-up work [117], the concurrency violations were clustered and filtered, facilitating fault localization of the root causes analysis for the developers, demonstrating its efficiency on the Floodlight controller. Indeed, the analysis of production-grade controllers presented in this chapter shows that concurrency issues are the root cause in many of the reported bugs in implementation of distributed protocols.

Empirical studies on distributed systems. Google's report on critical network outages [63], showed that control plane issues prevail in their SDN-based B4 WAN. Their analysis showed that *maintaining globally consistent network state* is a challenge, due to the *control plane convergence delays*, *inconsistency between control plane elements*, as well as *synchronization between data and control plane*. A number of partial and complete *failures of control plane elements and the control plane network*, including the *cascade of control-plane element failures* were observed. Noteworthy are also the *operational issues* due to the *buggy control plane software update push*.

Another empirical study on defects in well-known distributed systems, such as Cassandra and HDFS [198], showed that *faulty error handling* was the cause of 95% of catastrophic failures. In most

of the cases the error handling code was either empty or incomplete, ignoring the local failure which then propagated to entire system, or was overreacting, allowing a minor failure to crash the entire system. The authors also noticed *resource leaks* and *incorrect performance* issues, which have not been analyzed before in the context of SDN.

New vulnerabilities, due to cyclic dependencies between the control and data planes in distributed SDN, are discussed in [199]. The authors demonstrate how *control plane network failures* may render the cluster down, even in the absence of partitions. Illustrative examples of the problems of *oscillating leaders* and *lost leadership* were also presented.

Large scale empirical studies on real-life incidents in Google [63] and Microsoft networks [149], IP Backbone [111] and data center networks [56, 66] provide valuable data to the industry and to researchers, exposing network vulnerabilities and suggesting preventive measures. However, a comprehensive study on network control software in SDN is still missing. To fill this gap, this chapter systematically analyzes two of the largest open source repositories (10k+ bugs) to locate the vulnerabilities in production-grade distributed controller platforms.

5.2.2 Model-based Studies on SDN Control Plane Dependability

The first dependability studies on the SDN control plane consider the controller as perfectly reliable, assuming only control path link failures [72], and distributing the controllers only for latency reasons. More recent studies [106, 157, 15] also accounted for the software failures. Despite the diversity and complexity of SDN control plane failures, most of the studies on SDN control plane dependability reduce the controller to a single failure mode, i.e., assuming it is either operational or non-operational.

More complex dependencies and interactions between the elements of a complex systems use the SRN. The models described using SRN modelling formalism can be directly translated to large CTMC. The SRN models for the interaction between SDN control and data plane have been proposed in [60, 121, 127, 128, 161]. Overall, an important limitation of the previous models is the assumption about the perfect failover between identical controller replicas. However, the analysis presented in this chapter shows that simple controller replication is ineffective, because of i) shared failures, e.g., semantic bug in path computation, ii) faulty error handling mechanisms, which may lead to a erroneous failover and cause a cascade of controller failures and iii) failures specific to distributed control plane implementations, such as a software bug in distributed consensus protocols. These inefficiencies are modelled as a common mode failure (i), and the coverage factor (ii) in system dependability literature, while the failures specific to distributed systems (iii) are typically neglected.

Indeed, the complexity of interaction between SDN controller replicas has been widely overlooked in the literature. The failure correlation due to control plane misconfiguration was discussed in [128], while Mendiratta et al. [121] also discussed the imperfect failover. The study by Gonzalez et al.[60] modelled the synchronization process between controller replicas, with the focus on the trade-off between consistency and performance. Sakic et al. [161] provided a realistic response time model of the Raft consensus algorithm under different failure rates, complementing it with the measurements from an ODL testbed. The SRN models proposed in this paper, aim to combine all failure modes of distributed SDN implementations, for a holistic assessment of system dependability.

Model-based dependability assurance based on SRN has been successfully applied to various communication systems, such as software defined backbone network [128], NFV-based virtualized core [61], VoIP system [186], IaaS cloud [55], as well as distributed consensus protocols, such Raft [161], Paxos BFT [171], and their application in permissioned block chain systems [173]. This chapter follows a similar approach to provide high fidelity models that account for all failure modes encountered in the issue repositories of distributed SDN platforms.

5.3 Overview of Distributed SDN Implementations with ONOS and ODL

In this section the basic concepts of distributed systems are presented, with the focus on distributed SDN control plane, and in particular, on ONOS and ODL controller implementations. The analysis is based primarily on the official code documentation, supported by the presentations by ONOS[2] and ODL[3] distributed system engineering teams.

5.3.1 A Primer on Distributed Control Plane in SDN

In practice, a cluster of multiple SDN controllers is deployed in order to provide high performance, scalability and high availability. The logically centralized control plane operation is possible due to distributed protocols, which take care of the coordination, knowledge dissemination and seamless failover between different controller replicas. Services provided by the SDN control plane should be unaware of the distributed control plane implementation. Fig. 5.2a illustrates how the separation of concerns and location transparency are implemented. In order to manage large-scale networks, network state is partitioned into smaller chunks, called *shards*. Provisioning of a fault-tolerant system requires data shards to be replicated on several nodes. The service requests affecting the particular shard are routed to the controller node, which is responsible for that shard. In the example in Fig. 5.2a, the request consists of updating a flow rule on a particular device. After the flow rule has been updated, the response is propagated to a network application requesting an update, as well as the shard replicas. The exact order between theses steps depends on the style of the replication.

Distributed systems may use different replication styles, illustrated in Fig. 5.2b, depending on the application requirements, and data access patterns. Shard replicas can be updated in a *strongly consistent* or *eventually consistent* manner. i) Consensus based protocols like Raft [139] provide strong consistency, requiring the majority of the replicas to acknowledge the update before it can be committed by the leader, and used to create the response to the client. ii) Gossip protocols provide eventual consistency, using the epidemic style of propagation, where random pairs of neighbours compare their version of the data, known as *anti-entropy*, and agree on an appropriate final state if concurrent updates have occurred, which is known as *reconciliation*. iii) Another style of replication is *primary-backup*. Primary-backup replication can be done in a synchronous manner, where replicas are updated before the response, or in asynchronous, where the data change is first persisted locally

[2]Thomas Vachushka: ONOS Distributed Core and Jordan Halterman: Distributed Systems in ONOS with Atomix 3: Architecture and Implementation

[3]Colin Dixon: Clustering in OpenDaylight, Robert Varga, Jan Medved: OpenDaylight Clustering: What's new in Boron, Moiz Raja, Tom Pantelis: MD-SAL Clustering Internals

and replicated after the response. The performance-consistency trade-off is balanced by choosing the number of replicas and replication flavour.

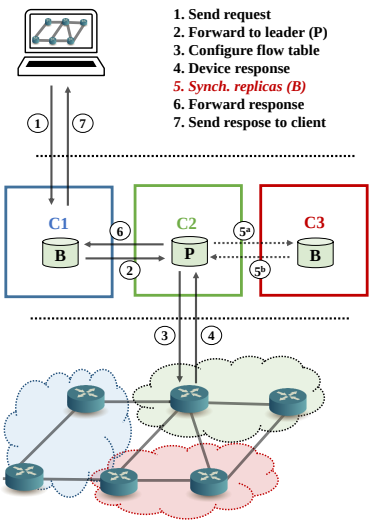
Timestamps and vector clocks are often used for ordering of the updates. Distributed systems are inherently asynchronous, execute independently and typically there is no global clock. Local clocks skew and drift, and even the Network Time Protocol (NTP) can provide an accuracy of tens of milliseconds. Ordering functions are necessary to enforce causal relationships between the events (e.g., port flapping). Hence, the vector clocks, also called *version vectors*, are often used instead. With Raft, the leader is responsible for the correct ordering of the updates.

Cluster membership and role management with Raft is illustrated in Fig. 5.2c, based on [139]. The controller (re-)joining the cluster starts as the *follower*. If the leader heartbeat is not received within a given threshold, it becomes a candidate, increases the election *term*, votes for itself and requests the votes from other members. Three outcomes are possible: if the candidate receives the majority of the votes before the *election timeout* it becomes the leader; if it discovers a candidate with higher term or a current leader it becomes the follower; otherwise, it increases the term count and starts the new election round. If a leader discovers a node with a higher term, it gives up the leadership. Raft uses randomized election timeout to reduce the probability of split votes. The leader election service can be used without using Raft's strongly consistent data replication, e.g., for the assignment of the primary-backup roles.

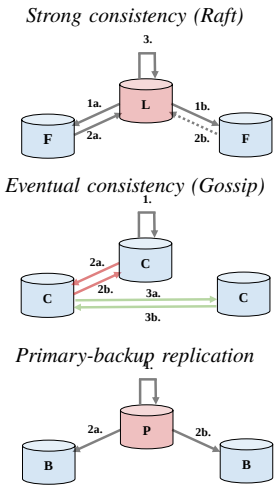
After leader failure, the follower with the largest term and the longest log will win the election. The leader proves its liveliness by sending periodic *heartbeats* to its followers. However, the independent controller nodes communicate over an unreliable network, which typically does not provide bounded delay guarantees, and it is practically impossible to distinguish between network and controller node failures. The messages can be delayed (network congestion or high load on the controller node), or lost (partitioned network or node crash), which can result in temporary inconsistencies between the network state seen by replicas. Failure detection is based on time thresholds, which have to be carefully tuned balancing the trade-off between stability and failure detection efficiency. The *φ-accrual failure detector* [71], is widely used in distributed systems, including in the SDN controllers implementations addressed here. The detector accounts for a suspicion level, $\Phi = -\log_{10}(1-F(t))$, where $F(t)$ represents a distribution of previous heartbeat inter-arrival times, implicitly assuming a normal distribution. Raft requires the majority of the controllers to be available, hence, it requires $2f+1$ controllers to tolerate f failures. In the case when network partitioning split the cluster into two parts that cannot communicate, either both partitions continue operating independently (favouring availability) or one of the partition freezes (favouring consistency), as consequence of the CAP theorem [144].

After a crash, a node re-joining the cluster has to catch-up with the rest of the cluster. All the changes to the data store are kept in a *journal*, or a *log*. *Log compaction*, or *state compression*, is the process of removing the entries from the log that no longer affect the current state. It is performed periodically to prevent the uncontrollable growth of the log. The shard replicas may request copies of a log entry from another replica in order to fill in missing transactions. Occasionally, a *snapshot* of data store state is saved, serving as a checkpoint in case the node crashes. Journals and snapshots are stored on disk for persistence.

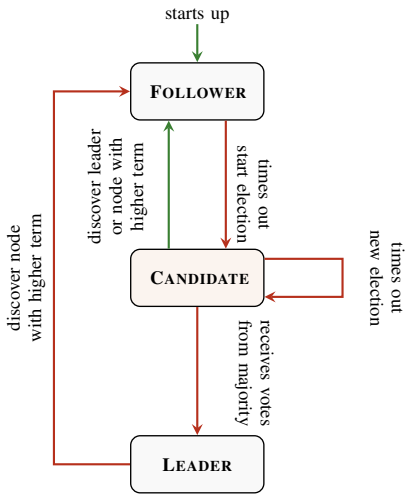
The implementation of distributed systems is non-trivial and requires fine-tuning of configuration parameters, as well as other operational issues that must be considered before the deployment. The



(a) Location transparency



(b) Data replication flavours



(c) Leader election with Raft [139]

Figure 5.2: A primer on distributed SDN implementations

controller nodes have finite resources, such as CPU and memory, which can easily exhaust if not planned and managed properly. The nodes may slow down during high load or computationally expensive operations, such serialization of large messages, or on persisting large chunks of data. Distributed implementations rely on 3rd-party libraries (e.g., Netty for low-level messaging and Kryo for serialization), which may introduce new vulnerabilities and interoperability issues, and have to be carefully tested. Moreover, the controllers are often deployed with virtualization platforms, whose configuration and dimensioning may affect the performance.

5.3.2 ONOS Implementation

The focus of ONOS has been on providing scalability, high availability and carrier-grade performance fulfilling the requirements of large operator networks. Hence, the distributed core was introduced from the beginning, and has evolved together with the application ecosystem.

ONOS provides low level distributed primitives, such as eventually consistent (`ECMap`) and strongly consistent (`ConsistentMap`), offering different consistency models and replication styles. Distributed primitives provide interfaces similar to standard Java classes, implementing the data structures and synchronization operations upon which data stores are built. Developer guidelines suggest that control plane data, such as resource reservation and other network configuration data, should use strong consistency. Data originating from the environment, such as network topology (read-intensive), should use eventual consistency to provide faster reaction to the network events. A primary-backup replication is used for the partitioned `FlowRuleStore`, while device mastership uses `LeadershipService` provided by Raft implementation.

The complexity of distributed protocols is encapsulated in Atomix [85], an event-driven Java framework for coordination of distributed systems, implementing a variety of different distributed protocols. It provides services such as cluster management, asynchronous messaging, leader election, data partitioning and replication. For scalable consistency, it introduces the concept of *group partitioning*, i.e., multiple Atomix clusters. Moreover, it offers a dynamic cluster membership, enabling the Atomix cluster to scale in or out based on the current demand.

5.3.3 ODL Implementation

ODL is a much larger and older project than ONOS, foreseen from the beginning to be "the Linux of the networks", supporting a variety of southbound protocols to ensure the smooth transition from legacy networks. The majority of ODL key partners are vendors, and the focus at the beginning was on the applications in data centers and coexistence with network virtualization technologies. The development of the major clustering project features started only after the fifth release (Boron).

ODL provides essentially two data stores, *configurational* to store a desired state, e.g. configuration of the flows, and *operational* store, storing the actual network state. All data is stored in a data tree, which is broken into shards. There are *module-based shards*, e.g., inventory, topology, while the rest of data goes to the default shard. The shards are replicated for high availability. Data replication uses the Raft consensus protocol, providing only a strong consistency model for all network primitives. The `EntityOwnership` service takes care of the leader election (shown in Fig. 5.2c), handles failover, and co-locates tasks and data. Data change notifications and Remote Procedure Calls (RPCs) operating on a given shard are directed to the entity owner, i.e., the leader. The nodes use RPC registry for localization of the entity owners. Gossip is used for the replication of RPC registry across the nodes.

In ODL, the Akka [84] framework encapsulates the complexity of the distributed protocols. *Akka actors* encapsulate the data tree shards, so interacting with the remote data shard is done by sending the messages to actors. *Akka clustering* implements Raft, and is responsible for the discovery of the nodes, their IP address, as well as the liveliness and reachability of the member. Cluster messaging relies on *Akka remoting*, while *Akka persistence* is responsible for durability.

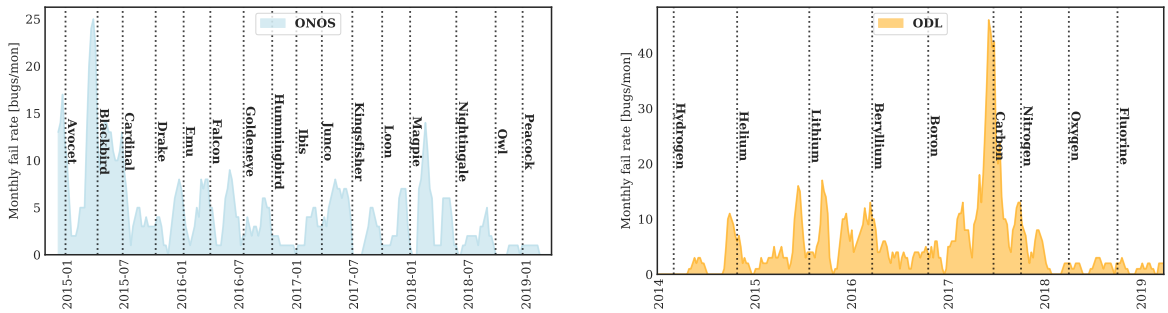
5.4 Localizing Dependability Bottlenecks in Distributed SDN Implementations

Next, the insights into defects reported in different functional areas of distributed control plane are provided, illustrated with a rich set of examples. The goal of has been to localize the most vulnerable components, and identify prevalent failure modes and their manifestation patterns.

5.4.1 Bug Repository

The problems in ONOS and ODL controllers are reported in their public Jira issue trackers[4]. Such bug repositories are a valuable source of information, as they contain the detailed fault reports from test and production environments. Only the issues labelled as "Bugs" are considered in this study, rather than new feature requests or enhancements. The issues related to defects in distributed implementations are extracted; in the case of ODL the issues tagged as part of the clustering project are selected, while in the case of ONOS manual inspection was used.

The number of bugs over time for both controllers is presented in Fig. 5.3. The monthly failure rate for ONOS peaks right before Blackbird (2nd release), while in the case of ODL the number of defects peaks before Carbon (6th release), which is consistent with these controllers' evolution.



(a) Open Networking Systems (ONOS)

(b) OpenDaylight (ODL)

Figure 5.3: The number of software defects related to distributed implementations reported over time for ONOS and ODL. The dates of major releases for both distributed controller platforms are indicated in the figure.

In total 500+ issues related to the distributed implementation have been reported. These issues are classified into the following four categories: i) defects in the implementation of distributed protocols (DP), ii) scalability and performance (SP), iii) high availability (HA), and iv) operational (OP) issues, as Illustrated in Fig. 5.4. Other issues are mainly requests for performance enhancements and code refactoring. Note that the categories are not mutually exclusive. In the case of ambiguity, the bug is assigned to the primary trigger. For example, when the topology state cannot converge after the node restart, is is assigned it to the HA category. As another example, when the slow controller node triggers false positives in the failure detector, the bug is assigned to the SP category. The references to the representative issues in each category are provided in the following sections.

[4]Data retrieved on March 3, 2019 from ONOS and ODL bug repositories

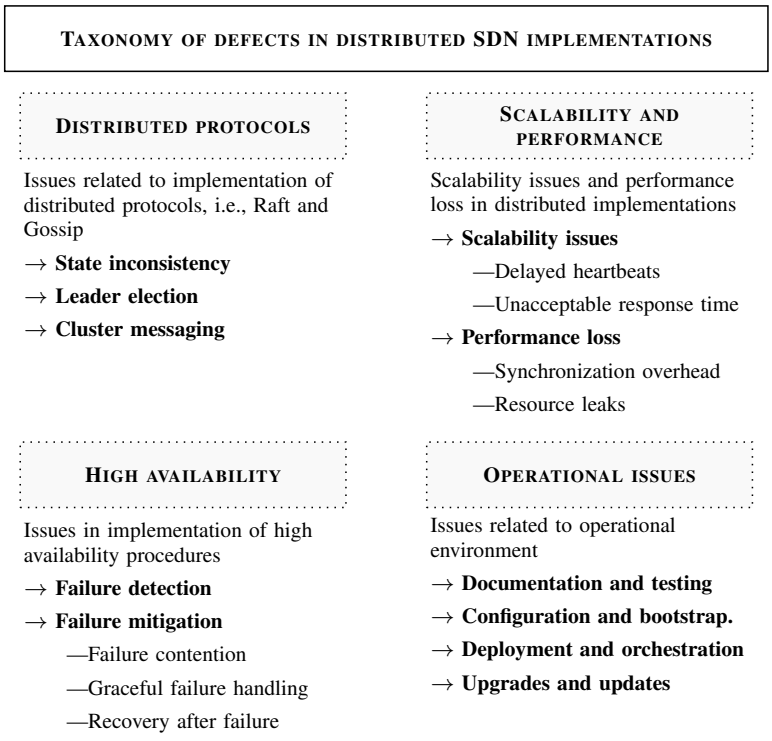


Figure 5.4: Taxonomy of defects in distributed SDN control plane implementations.

5.4.2 Defects in the Implementation of Distributed Protocols (DP)

In a multi-controller architecture, all controllers must have a consistent view of the network state in order to provide correct logically centralized operation, which is ensured by means of distributed protocols, such as Raft and Gossip. In total, 216 (40%) issues have been identified in this category, related to state inconsistency, leader election process, and cluster messaging implementations.

5.4.2.1 State Inconsistency

State inconsistency between control plane and data plane elements has already been identified as one of the leading causes of critical outages in operational SDN networks [63]. The analysis affirms this finding, discovering that 52(10%) of defects have been reported in this category.

Reported issues include different cases, such as when only the master of devices sees correct flow count [dp1], or when the links disappear after load balancing [dp2].

One reason of the state inconsistency is the *missing data change notifications*. For example, some `DEVICE_ADDED` events were not replicated to all instances in the cluster [dp3], which led to a large increase in switch-up discovery, since without synchronous replication, the instances need to wait for Gossip to synchronize on forwarding device events. As another example, it was noted that some `ConsistentMap` operations, such as put and replace, do not publish data change notifications [dp4]. Sometimes the notifications are missing only in certain conditions, e.g., some links disappear when a

forwarding device changes master [dp5]. After restarting, ONOS nodes learn about the topology via Gossip, but it happened that the hosts sometimes have missing IP [dp6].

The second root cause of the state inconstancy are *cluster synchronization issues*. It was noted that in the relaxed consistency mode (strong consistency where reads are performed from a local cache instead from the leader), it is possible to be out-of-sync indefinitely [dp7], while [dp8] reports that the node re-joining as follower could not synchronize, and the lagging follower had to be forced by the leader to install a snapshot.

Several bugs regarding the *event ordering issues* were also reported. One issue exposes the problem with `RaftActor`, as the last applied index moves backwards, leading to violation of transaction ordering [dp9]. Another example of potential order violation is reported in [dp10], when time moves backwards due to the Daylight Saving Time, suggesting that instead of calendar, the vector time should be used for versioning.

5.4.2.2 Leader Election Issues

Leadership assignment and hand-off are essential for load balancing, scale-in/out and failure mitigation operations. Our findings show that a stable leader/master is hard to implement, given that 58(10%) issues were reported in this category.

Misbehaviours, such as `EntityOwnership` least load policy not working as expected [dp11], and not balancing the load properly after the nodes are restarted or isolated from the network several times [dp12], have been reported. Sometimes, the controller role change messages are not being delivered [dp12,dp13]. In another representative example, the devices intermittently loose their master, caused possibly by Atomix `AppendRequest` timeout failure [dp14].

5.4.2.3 Cluster Messaging System

Another challenge lies in the implementation of the reliable cluster messaging system, which relies on 3rd-party data serialization (Kryo) and messaging (Netty) libraries. Serialization is an expensive operation which can significantly slow down the controller, degrading the performance and eventually leading to the crash of other operations. A BGP router crashing during Kryo serialization was reported in [dp15], while in [dp16] the processing of a large message triggered incorrectly `UnreachableMember`. The default failure-detector (ODL) triggers if there are no heartbeats within 5.5s in Akka cluster, and if the serialization time exceeded this time threshold, a false alarm would occur.

5.4.3 Scalability and Performance (SP) Issues

Increasing the scalability of the control plane should not affect the system performance [140], which should remain stable over the long hours of operation. However, 91 (17%) issues belonging to this category have been identified.

5.4.3.1 Scalability Issues

Providing a performant SDN control plane is non-trivial for large service provider networks, which induces high load on the controllers, both, in terms of topology size and volume of network events they must handle.

A recurring issue in both controllers is seen when processing a large number of events, which slows down a node, *delaying the heartbeats*. Several issues related to the unexpected `UnreachableMember` when the cluster is under load have been reported, under un umbrella bug [sp1]. This happens when cluster members have multiple (or big) messages to process, and they are late to read heartbeats from peers. Delayed heartbeats have severe consequences on the cluster operation, leading to frequent leader re-election, control plane instability and eventual crash, as discussed in [68] and [44]. Indeed, Raft requirements for the correct leader election and stable operation requires the following constraints to be satisfied [139]:

$$\textit{BroadcastTime} \ll \texttt{electionTimeout} \ll \textit{MTBF}$$

where *BroadcastTime* represents the time to send to and receive responses from all the cluster nodes in parallel (including network propagation delay and node processing time), *MTBF* is the mean time between failures of a single server, and `electionTimeout` is a timeout triggering re-elections, illustrated in Fig. 5.2c.

Slow controller under load can also cause other operations to timeout, such as the installation of a large number of intents, which exceed the hard coded timeout of 5 s for processing and installation [sp2]. Other operations were reported to misbehave under load, as well. For instance, balancing of the masters not working properly with a 625-switch network [sp3], and a large Chordal topology with 300 links not converging until the node restart [sp4]. Furthermore, potential data loss during scale-out operation [sp5] was also reported.

5.4.3.2 Performance Regression

Maintaining the same performance at scale is presumably an even harder challenge, due to the overhead introduced by distributed protocols (e.g., leader election, consensus-based replication), as well as the resource leaks, which can degrade the performance over time.

Several issues related to *performance overhead and regression* in multi-node setup have been reported, e.g., maximum number of installed intents being lower in the cluster than in the stand-alone mode [sp6]. Unsatisfactory performance is also frequently reported, such as the resource reservation taking too long in multi-node cluster [sp7], resulting in an unacceptable reaction time to network topology events. The performance overhead in a cluster setup when using strongly consistent Raft replication style was discussed in [161], and in [160] the authors proposed adaptive consistency models to balance response time and reliability.

Resource leaks, such as unclosed transactions and memory leaks, can cause the performance to degrade over time and the controller crash, due to the resource exhaustion. A bug in the Atomix log compaction timer [sp8] caused the nodes to eventually run out of disk space, by being filled up with Raft commit logs. As another example, the resource store opened a new Raft session on each transaction [sp9], due to a change in session management after the Atomix upgrade. Moreover, a number of memory leaks have been reported, related to flow rule [sp10], distributed meter store [sp11], Netty messaging manager [sp12], and Kryo serialization [sp13]. Note that the increase in resource consumption does not happen only due to the bugs, but also as a design choice. For instance, expired flows (i.e., flows with idle timeouts) remain in the ODL *configurational* data store [sp14], while in

ONOS `EventuallyConsistentMap` naturally grows due to the usage of tombstones replacing dead objects [sp15].

5.4.4 High Availability (HA) Issues

HA is a key enabler for mission critical operations, and in many use cases the main reason to adopt a distributed SDN design. The principles to ensure HA are reliable failure detection, failure contention, and fast recovery. Nevertheless, the analysis exposed 118(21%) defects in HA subsystem.

5.4.4.1 Failure Detection

Failure detection in ONOS and ODL is based on the φ-accrual failure detector [71], which detects when the heartbeat intervals have exceeded a given suspicion level. The parameters of the failure detector should be carefully tuned to avoid false positives, and subsequent cluster destabilization, which may trigger unnecessary leader re-elections. The `heartbeatInterval` and `phiFailureThreshold` could not be configured in the first releases [ha1]. In addition to false positives caused by slow-performing nodes, which delay heartbeats, some other events can trigger false alarms. For instance, when the controller node is being added to a cluster [ha2], or when the configuration change is applied to the data store [ha3], unnecessary state changes occurred.

5.4.4.2 Failure Mitigation

When actual failures do happen, it is important that nodes fail gracefully, without destabilizing the cluster, and reload quickly, recovering the state and catching up with the rest of the cluster.

Failure contention is not trivial, due to a tight interaction and interleaving of the cluster members. Failure contention mechanisms, that should be in place to avoid that a failure of one instance brings down the entire cluster, can be buggy themselves [ha4, ha5, ha6]. An example of such cascade of failures caused by the false positives in failure detector triggering frequent leader re-election has been already mentioned in the previous section. Another representative example of a mishandled failure contention is a positive loop, reported in [ha7]. In the case of a high-load scenario, a Raft client continually retried a failed operation as long as it could maintain its session, increasing CPU/memory usage in already overloaded partitions, causing the cluster to spiral out of control.

Failing fast and gracefully is another desirable property of highly-available systems. Reports showed that when ONOS gets an exception such as out of memory, it hangs in a non-recoverable state, instead of crashing hard [ha8]. In another example, a fast fail feature was requested when `DatabaseManager` does not start cleanly, given that many other subsystems have transitive dependency to this module [ha9]. If a leader of a given shard/partition, the leadership handover should happen quickly [ha10,ha11] and without a data loss [ha12,ha13].

An efficient recovery after failures: Snapshots of data stores and transaction journals are occasionally persisted for durability, to ensure quick recovery after failures. The state persistence is not perfect, as reported in [ha14,ha15], for flow and intent stores in ONOS. Nevertheless, the most prevalent issue is faulty recovery, with 53(10%) reported bugs. Typical issues are a node failing to join and sync with the rest of the cluster [ha16,ha17], and a data loss upon restoration [ha18,ha19], leading to the state inconsistency between controller replicas.

5.4.5 Operational (OP) Issues

Operational issues include supporting functions, not necessarily related to the buggy controller code, but rather to practical deployment scenarios. This category includes 76(14%) issues, related to documentation and test automation/coverage, cluster configuration and bootstrapping, interworking with virtualization platforms, upgrades and updates of the 3rd-party libraries.

5.4.5.1 Documentation and Testing

An adequate documentation should be provided to facilitate correct usage and configuration of the multi-node cluster [op1]. The execution of the test suites should be automated [op2], and occasionally extended with the new test cases [op3,op4], covering new failure modes, which were previously unaccounted for.

5.4.5.2 Cluster Configuration and Bootstrapping

The controllers in a cluster have to be correctly setup and able to automatically discover the peers. Before joining the cluster, the controller software has to be correctly loaded and configured. Both controllers use Apache Karaf, an open source implementation of OSGi framework, to automatically load and manage individual controller bundles (i.e. modular services, built-in applications). The issues, such as serialization of cluster configuration change [op5] and Karaf issues causing some bundles not to load in 3-node cluster [op6], are typically discovered before the deployment.

5.4.5.3 Deployment & Orchestration Issues

The controller software requires a host operating system, and the multi-instance setup is often deployed in virtualization environment, with dockers or virtual machines. The interactions with virtualization layers, and corresponding orchestration tools, e.g., Kubernetes, have to be carefully tested [op7, op8,op9].

5.4.5.4 Upgrades and Updates

The operation of ONOS and ODL relies on many 3rd-party libraries, such as Kryo and Netty, which are separately developed and maintained. The regression tests must be in place to efficiently detect 3rd-party vulnerabilities and compatibility issues. The same is valid for internal modules, which are not immune to performance regression and backward compatibility issues. For instance, a throughput regression was reported after an upgrade to Atomix 3 [op10].

5.4.6 Prevalent Failure Modes

The percentages of bugs of each presented categories have been summarized in Tab. 5.1. The most vulnerable category is the buggy implementation of the distributed protocols (40%), in particular maintaining of the consistent state and stable leader/mastership election. The second source of defects lies in the implementation of HA mechanism (21%), mainly due to failure contention and imperfect recovery. Performance and scalability issues are the third most prevalent (17%) root cause of the bugs, primarily due to the hard coded timeouts and resource leaks.

Table 5.1: Distribution of software defects by category: distributed protocols (*DP*), scalability and performance (*SP*), high-availability (*HA*) and operational (*OP*) issues.

Category	ONOS	ODL	Total
DP	97 (44%)	119 (36%)	216 (40%)
SP	39 (18%)	52 (16%)	91 (17%)
HA	42 (19%)	76 (23%)	118 (21%)
OP	30 (14%)	46 (14%)	76 (14%)
Other	13 (6%)	42 (13%)	55 (9%)
Total	221 (40%)	335 (60%)	**556**

The presented categories of defects significantly differ in their impact on the network services. Bugs in the implementation of distributed protocols, leading to the state inconsistency between control and data plane, may not interrupt the control plane operation, but are nevertheless critical for the network. Having a stale network topology view may lead to an installation of paths with loops (congestion), blackholes (packet loss), or even forbidden paths (violation of security policy) [63]. Performance degradation over time, due to memory leaks, can slow down the controller node [125], leading to a delay of the heartbeats, which consequently may trigger a fatal cascade of control plane failures [44, 63]. On another hand, operational issues, although critical, will typically not be perceived by a user, once the network is successfully deployed in the production environment.

5.5 Modelling Abstractions for Imperfect Distributed SDN Implementations

Next, the comprehensive models for imperfect distributed control plane are provided, accounting for different failure modes and their manifestation patterns, characterized in the previous section. The modelling abstractions for imperfect distributed SDN implementations are provided in the formalism of SRN, a stochastic extension of Petri Nets [179]. The key SRN modelling ideas are shown in the examples of cluster (Fig. 5.5), service request (Fig. 5.6) and preventive maintenance (Fig. 5.7) models.

In the SRN framework, the combination of markings in the *places* (circles) represents model states. The system state may change upon firing of the *activities*, which can be instantaneous transition (`inconsistent_state` → `sw_ok`), or follow a deterministic (`sw_prone` → `planned_restart`), or an n.e.d distribution. An SRN model can be automatically translated to equivalent CTMC. The states and activities are associated with the corresponding rewards, which allows straightforward evaluation of system performance metrics, such as the expected number of operational controllers.

Similar to guard functions of the SRN framework, Stochastic Activity Networks (SAN), have the concept of gates, allowing for compact model representations. *Input gates* (red triangles) define enabling predicates, i.e., a necessary precondition for triggering of the activity. For instance, in Fig. 5.6, a request can be served only if the majority of the controllers are operational. *Output gates* (black triangles) define more complex state change upon the activity firing, e.g., after cluster recovery all the controller are reset to `sw_ok` state.

Closed form solution can be obtained for simpler small size problems with exponential and instantaneous transitions. For large problem instances with general transition time distributions, either

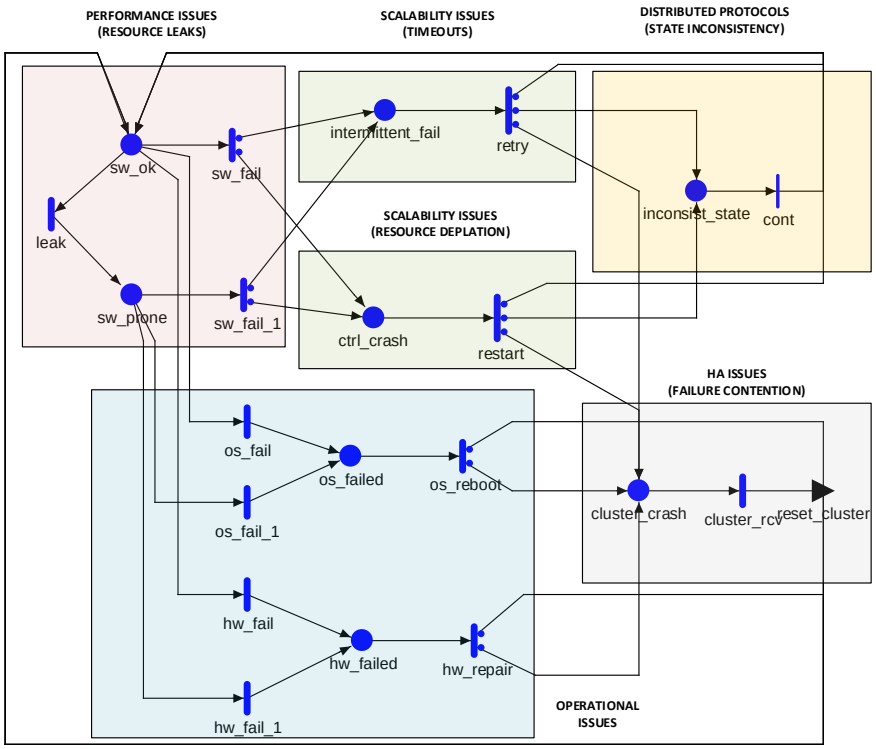


Figure 5.5: Modelling abstraction for imperfect SDN cluster.

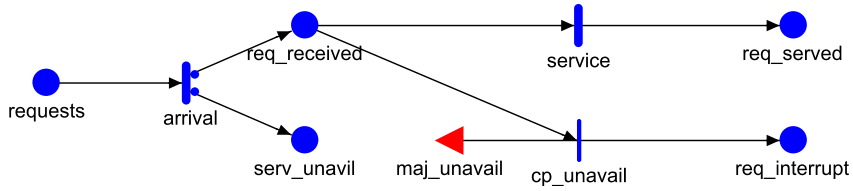


Figure 5.6: SRN for service request dynamics.

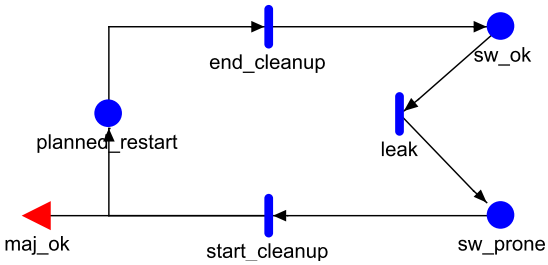


Figure 5.7: SRN extension for preventive maintenance, i.e., software rejuvenation.

numerical solution or discrete event simulation is used. Solvers, such as Mobius [42], SPNP [75] and SHARPE [74] can be used for this purpose.

5.5.1 Modelling Abstraction for Imperfect SDN Cluster

The failure modes, i.e., defects in distributed protocol implementations, scalability and performance, high-availability, and operational issues, discussed in Sec. 5.4, are incorporated in the proposed SRN model in Fig. 5.5.

5.5.1.1 Resource Leaks

When the controller is initiated or reloaded after a crash, it starts from the clean state `sw_ok`. The baseline software failure rate in this state is λ_0. During the continuous operation, the resource leaks are accumulated and the controller performance starts to degrade. A common way to model this effect is to assume that the risk of failure significantly increases after a certain utilization threshold is exceeded [80], as seen in practice [sp15]. For instance, [44] note that the controller throughput and response time degrade significantly when available memory is below 4 GB. This effect is modelled by introducing the state `sw_prone`. The time to reach this utilization threshold depends on the controller load and the type of network applications serves. The randomness in the resource leak process is accounted for by modelling it as Poisson process with the rate λ_{leak}.

5.5.1.2 Soft and Hard Software Failures

The analysis in Sec. 5.4 showed two distinct types of failure effects: soft failures, resolved by a simple retry of the operation (`intermittent_fail`); and hard failures: leading to a controller crash, requiring a restart (`ctrl_crash`). *Soft failures* are short interruptions in the controller operation, due to failed or timed-out transactions, concurrency and data race issues, leader movement and load balancing. They are typically resolved by retrying the operation, which occurs at the rate μ_{retry}. *Hard failures*, i.e., software crash, can happen when the controller node runs out of resources, e.g., with out-of-memory error. The controller restart time (rate) $\mu_{restart}$ accounts not only for the application restart, but also for the loading time of all dependent bundles, as well as the time to reconnect with peers, and discover the leaders of data store shards.

5.5.1.3 Transient State Inconsistencies and Cluster Crash

A qualitative analysis in the previous section showed that software failures can either be either: i) successfully recovered from, or ii) can result in transient state inconsistencies, or iii) cluster-wide failure. *Transient state inconsistencies* (`inconsist_state`) occur with probability p_{state}, due to missing notifications, wrong event ordering, lagging followers, data loss in journals and snapshots, which are used upon recovery. *Cluster failures* (`cluster_crash`) reflect the cases when the failure contention fails (with probability p_{crash}) and crash of a single controller brings down the entire control plane. The cluster repair rate $\mu_{cluster}$ is longer than the restart of a single controller node.

The distribution between successful repairs (p_{ok}), transient state inconsistencies (p_{state}) and cluster crash (p_{crash}) is hard to estimate from the bug reports. Hence, we make a reasonable assumption, and conduct the sensitivity analysis to evaluate the impact of uncertainty regarding its value.

5.5.1.4 Operational Failures

From all operational issues discussed in Sec. 5.4.5, the interaction with the environment, e.g., host operating system (`os_failed`) and computing hardware (`hw_failed`) will have the highest impact on the services in the production environment. The operating system, including the virtualization layer, fails with the rate λ_{os}, and is rebooted with the rate μ_{os}. Similarly, computing hardware fails with the rate λ_{hw}, and is repaired with the rate μ_{hw}. Bugs in the failure contention mechanism may lead to a complete system crash, which happens in p^{os}_{crash} and p^{hw}_{crash} of the cases, respectively.

A common failure mode is introduced, in order to account for different deployment scenarios. In cases when controller replicas are deployed as virtual machine (VM) on the same server, the crash of a server will render all the replicas down. Similarly, in case when the replicas run in Docker container (DC), the host operating system is shared as well, and its failure will lead to a cluster-wide failure. This effect is modelled by adding `reset_cluster` output gate following the transitions `{os,hw}_fail`$_{1,2}$ (not included in Fig. 5.5).

5.5.2 Reference Stand-alone Model

A controller operating in a stand-alone mode is used as a reference model, to evaluate the gains in terms of control plane dependability in a distributed setup. In the stand-alone mode, many of the failure modes will be shared, since the controller uses the same data structures as in the cluster mode. State inconsistencies between control and data plane can still occur, due to the slow node failing to process the network events on time, or faulty journal recovery upon restart. Resource leaks, especially those related to the natural increase in memory usage, do occur as well in the stand-alone operation. Hence, the model in Fig. 5.5 is modified, by removing `cluster_crash` place, and all the transitions associated with it ($p_{crash} \rightarrow 0$).

5.5.3 Modelling Abstraction for Control Plane Services

SDN controllers provide services, such as the management of the forwarding devices through the south-bound interface, and the implementation of high level policies through the north-bound interface. The generic modelling abstractions for control plane services is illustrated in Fig.5.6. A given number of requests $N^{\text{req}}_{\text{sent}}$ arrives with a given rate λ_{req}, and are served at the rate μ_{req}. In cases when the majority of the controllers is down, new requests cannot be processed ($N^{\text{req}}_{\text{unavail.}}$), and the ongoing requests will be interrupted ($N^{\text{req}}_{\text{interrupt.}}$). λ_{req} and μ_{req} depend on a particular service, as well as the performance of the particular control plane configuration. The serving rate can be tied to the number of operational controllers and current simulation time, accounting for a degraded performance due to resource leaks. For the simplicity, μ_{req} is kept constant throughout the study, leaving it to the future work to study more complex parameter relationships.

5.5.4 Preventive Maintenance Policies

The failure rate after long hours of operation (`sw_prone`) is higher due to the lower amount of available resources, caused by resource leaks, i.e., *software ageing* [80]. An operator can decide to preventively restart the controller, cleaning up the internal data structures, dead objects, zombie processes, and unclosed connections. Such preventive measure can be implemented by starting a timer (deterministic action with rate λ_R), once the certain utilization threshold is reached. Duration of the planned outage

($1/\mu_R$), also called *software rejuvenation*, of the controller depends on the level of rejuvenation, e.g., process or application restart. We assume rejuvenation is triggered only when majority of controllers is available.

5.5.5 Dependability Metrics of Interest

Dependability metrics are defined by assigning rewards. The metrics of interest are: i) the Steady State Availability (SSA), ii) failure dynamics, and iii) user-perceived service availability.

5.5.5.1 Steady State Availability (SSA)

The SSA is evaluated as the probability of being in the operational states: `sw_ok` and `sw_prone`. Note that the place related to state inconsistency (`sw_state`) is a transient state, since it is followed by an instantaneous transition (`cont`), and hence the state is used to count occurrences of state inconsistencies.

Depending on the replication style, the control plane will need the majority of the nodes participating in the cluster to be available (Raft), at least two operational controllers for primary-backup replication, or at least one operational replica for Gossip style replication. The model does not differentiate leader and followers, since different nodes may be leaders for different data shards. We define availability metrics as:

$$A_i = \begin{cases} A_{\texttt{maj.}} = P\{N_{\text{operational}} > \lfloor \frac{\text{N}}{2} \rfloor\} & (\textit{Raft}) \\ A_{2/\text{N}} = P\{N_{\text{operational}} \geq 2\} & (\textit{P-B}) \\ A_{1/\text{N}} = P\{N_{\text{operational}} \geq 1\} & (\textit{Gossip}) \end{cases} \tag{5.1}$$

where the number of operational controllers is defined as:

$$N_{\text{operational}} = \texttt{Tokens(sw\_ok)} + \texttt{Tokens(sw\_prone)}$$

5.5.5.2 Failure Dynamics

The time spent in individual failure states (rate reward) indicates the contribution of different failure modes to control plane outages, as well as the frequency (impulse rewards) of different controller and system failures.

5.5.5.3 User-perceived Service Availability

Depending on the service, the control plane availability will be sampled at different times, i.e., at request arrival, and for a different duration, i.e., request serving. Since the network operator is interested primarily in the user-perceived service availability, the control plane availability is sampled only when the service is active. The following metrics of interest are defined as *Service Availability* (*SA*), *Service Continuity* (*SC*) and *Request Completion Success Probability* (*SR*):

$$SA = \frac{N^{\text{req}}_{\text{received}}}{N^{\text{req}}_{\text{sent}}} = \frac{N^{\text{req}}_{\text{received}}}{N^{\text{req}}_{\text{received}} + N^{\text{serv.}}_{\text{unavail.}}} \tag{5.2}$$

$$SC = \frac{N^{\text{req}}_{\text{served}}}{N^{\text{req}}_{\text{received}}} = \frac{N^{\text{req}}_{\text{served}}}{N^{\text{req}}_{\text{served}} + N^{\text{req}}_{\text{interrupt.}}} \tag{5.3}$$

$$SR = \frac{N^{\text{req}}_{\text{served}}}{N^{\text{req}}_{\text{sent}}} = \frac{N^{\text{req}}_{\text{served}}}{N^{\text{req}}_{\text{received}} + N^{\text{serv.}}_{\text{unavail.}}} = SA \times SC \tag{5.4}$$

5.6 Characterization of SSA, Failure Dynamics and User-Perceived Service Availability

Next, the case study on realistic SDN controller platforms is presented. The proposed models are used to quantify control plane dependability metrics. Moreover, the practical applications for network operators are shown, by analysing different deployment scenarios and preventive maintenance policies.

Model parameters are based on empirical data presented in Sec. 5.4, and on the studies of software components of a similar complexity. Parameters related to the software failure rates [127, 185], resource leaks [190], and recovery procedures [120, 185], as well as the parameters related to the availability of operating system and computing hardware [94, 127], are presented in Tab. 5.2.

Table 5.2: SRN model parameters [34, 120, 127, 185, 190]

Parameter	Description	Baseline value
λ_0^{-1}	Baseline failure rate (`sw_ok`)	7 days
λ_{high}^{-1}	High failure rate (`sw_prone`)	3 days
λ_{leak}^{-1}	Resource leak rate	1 day
μ_{retry}^{-1}	Retry operation upon timeout	5 sec
$\mu_{restart}^{-1}$	Application process restart	3 min
$\mu_{cluster}^{-1}$	Restarting cluster nodes	10 min
p_{soft}^{ok}	Proportion of soft failures (`sw_ok`)	0.75
p_{soft}^{prone}	Prop. of soft failures (`sw_prone`)	0.25
$p_{hard}^{ok,high}$	Prop. of hard failures	$1 - p_{soft}^{ok,high}$
p_{ok}	Probability of successful recovery	$1 - p_{st.} - p_{cr.}$
p_{state}	Prob. of transient state inconsistency	0.40
p_{crash}	Prob. of cluster-wide crash	0.05
λ_{os}^{-1}	Mean time between OS failures	60 days
μ_{os}^{-1}	OS reboot time	30 min
λ_{hw}^{-1}	Mean time between HW failures	6 months
μ_{hw}^{-1}	HW replace time	2 hours

5.6.1 Control plane availability

5.6.1.1 SSA

The SSA for different cluster configurations is presented in Tab. 5.3. Since the the availability for larger clusters are rather small, the Steady State Unavailability (SSU) is presented instead, in order to illustrate the magnitude of difference between various cluster configurations. We observe that unavailability of stand-alone controller is an order of magnitude higher than in distributed setup (N>1), because of better fault tolerance to the failures of single controller instance. However, as the number of controllers in the cluster increases, the unavailability of the cluster actually slightly decreases. This effect is in part due to specific, cluster-induced, failures, such as the ones due to faulty failure contention. Other reason why the unavailability of strongly consistent application is lower in larger clusters, is due to the fact that larger number of the cluster members (i.e., majority) is required to be operational.

Table 5.3: Steady State Unavailability (1 − SSA)

Unavailability	N=1	N=3	N=5	N=7
1-$A_{1/N}$	1.2176e-03	3.1460e-04	5.1909e-04	7.197700e-04
1-$A_{2/N}$	1.0	3.1899e-04	5.1909e-04	7.197700e-04
1-$A_{maj.}$	-	3.1899e-04	5.1911e-04	7.197701e-04

5.6.1.2 Sensitivity Analysis

The sensitivity analysis for $A_{2/3}$ is conducted to study the impact model parameters uncertainty.

It can be observed in Fig. 5.8 that cluster recovery failures $\mu_{cluster}$ and failure contention success probability $1 - p_{crash}$ have the largest impact on availability of strongly consistent services $A_{2/3}$. The qualitative analysis in the previous section exposed many defects in failure contention mechanism. The results of the sensitivity analysis only emphasize the need to prioritize the hardening of failure contention mechanism.

The following parameters, by the impact of their uncertainty of the strongly consistent services are failure rate in failure-prone state λ_{high} and resource leak rate λ_{leak}. Unfortunately, these parameters depend on many factors, such as workload, service request type, hardware configuration, available resources (CPU, RAM, etc.), and hence, have to be measured for a particular distributed setup and use case.

The uncertainty of software λ_0, operating system λ_{os} and hardware λ_{hw} failure rates has slightly lower impact. Fortunately, these parameters are well reported in the past empirical and model-based studies.

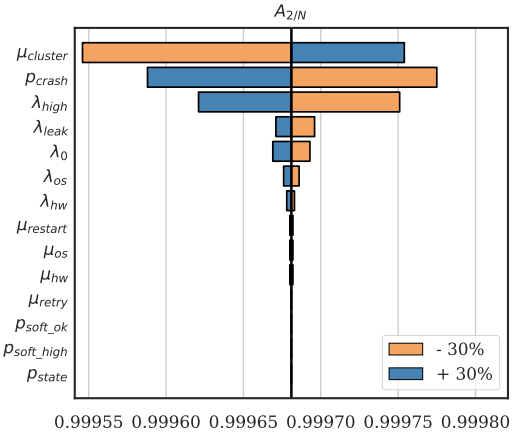


Figure 5.8: Sensitivity analysis for $A_{2/3}$.

5.6.2 Failure Dynamics

The failure modes differ significantly in terms of their frequency and control plane outage, which we define here as an event in which the majority of the controllers were unavailable.

The expected control plane outages within one year of operation of 3-node cluster is 10.25, with a cumulative duration of 2.8 hours. In 96% of the cases the control plane failure is caused by unsuccessful failure contention. Moreover, the state inconsistency between control and data plane elements is expected to occur on average 73.91 times within one year. Although transient state inconsistencies are resolved quickly, not affecting the control plane availability, they can have adverse effect on data plane operation. State inconsistency can cause traffic loss by installing the paths with blackholes, or overload the links by installing the paths with loops, and install flow rules implementing conflicting policies. Practical experience reports on operational SDN networks demonstrate that state consistency issues cannot be neglected [63].

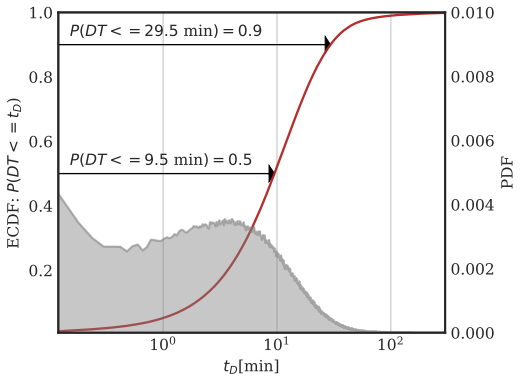


Figure 5.9: Downtime (DT) distribution.

The downtime distribution (ECDF) is presented in Fig. 5.9. As a reference, PDF is also presented (shaded grey area). We observe that the median of system outage in a reference setup of 3-node cluster duration is below 10 min, while 90%-tile is below 30 min, with many short-term interruptions due software failures. Such failure dynamics of the control plane failures has a detrimental impact on which services get affected by the system outages.

5.6.3 User-perceived Service Availability

User-perceived availability depends on the service dynamics, i.e., request arrival (λ_{req}) and serving rates (μ_{req}). The impact of λ_{req} and μ_{req} on service availability metrics, *SA*, *SC* and *SR*, is presented in Tab. 5.4 and Fig. 5.10. In Tab. 5.4 several typical services are presented. Request serving rate ranges from 500 ms for an installation of large batch of flows, up to 15 min for in-service software upgrades (ISSU). Request arrival rate varies between 1 min to 1 hour, representing different control plane traffic patterns, e.g., `PACKET_IN` or network statistics poll. It can be observed that $SR = SA \times SC$ is mainly affected by *SA*, service unavailability at the moment of request arrival, more than service continuity *SC*, which is an order of magnitude higher in a given setup.

Fig. 5.10, illustrating unsuccessful service request completion rate ($1 - SR$), demonstrates how the longer serving rate can increase the service unavailability up to an order of magnitude. Similarly, higher request arrival rate, resulting in frequent sampling of the control plane availability, results in

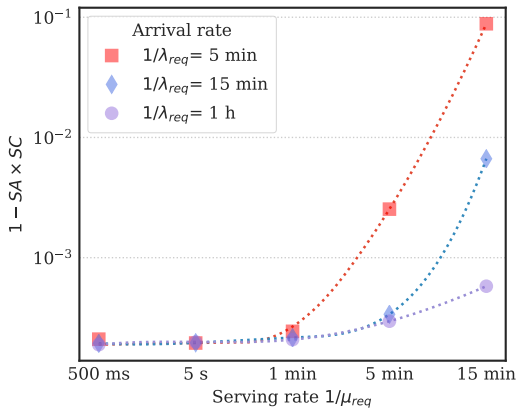


Figure 5.10: Request Completion Failure Rate (1-SR).

Table 5.4: Service request and serving rates

Service	Service A	Service B	Service C	Service D
λ_{req}	5 min	5 min	1 h	1 h
μ_{req}	500 ms	5 sec	5 sec	5 min
SA	99.9790%	99.9807%	99.9802%	99.9814%
SC	99.9999%	99.9998%	99.9999%	99.9890%
SR	99.9790%	99.9805%	99.9801%	99.9704%

lower user-perceived service availability, as it is more likely to be affected by short, but frequent software failures.

5.6.4 Comparison of Different Deployment Scenarios

In small resource-constrained networks, such as industrial networks [7], the network operator may choose to run the cluster of controller nodes on shared physical machines. Deploying the controllers in separate VM provides better isolation between software instances, but introduces additional overhead, since every instance runs its own operating system. DCs provide a lightweight virtualization, but imply an additional common mode failure, since a crash of the operating system will render all instances unavailable.

Control plane availability $A_{2/N}$ for different deployment scenarios is illustrated in Fig. 5.11. It can be observed that in the case of VMs running on the same physical machine, i.e., shared hardware failures, the availability is only slightly lower. In the case of DC, the availability loss is much higher, being an order of magnitude lower than in the first two deployment scenarios.

5.6.5 Optimization of the Preventive Maintenance Policies

The impact of different rejuvenation policies, i.e., rejuvenation scheduling λ_R, for different rejuvenation duration is illustrated in Fig. 5.12. It can be observed that early rejuvenation is beneficial in all studied scenarios, and should be done in the optimal case as soon as the `sw_prone` state is entered ($\lambda_R \rightarrow \infty$). The operator and/or controller software developers should determine the precise threshold when this state is reached, by measuring the resource leak rates (λ_{leak}) for a given configuration setup and

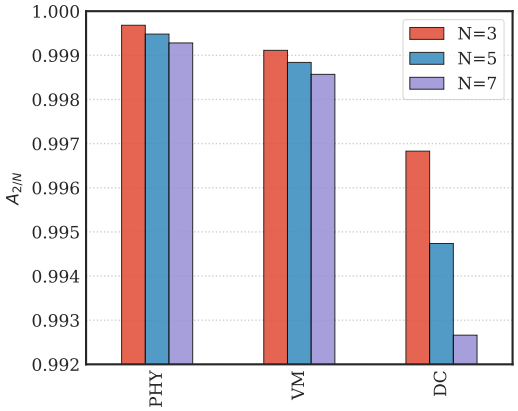


Figure 5.11: $A_{2/N}$ in different deployment scenarios: separate physical machines (PHY), virtual machines (VM) and docker container (DC) sharing the same physical hardware.

operational workload profiles. An exhaustive measurement campaign for real-life distributed SDN controller platforms is conducted is discussed in the following chapter.

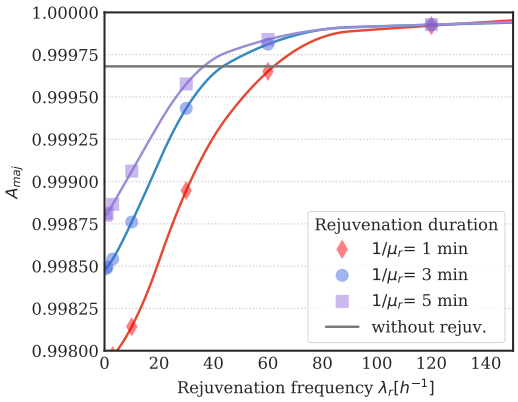


Figure 5.12: Software rejuvenation policies.

5.7 Concluding Remarks

5.7.1 Summary

This chapter presents a comprehensive analysis of defects and vulnerabilities in real-life SDN controller platforms, as well as the modelling abstractions of imperfect distributed controller plane. The analysis in the first part has demonstrated that, although some particular defects have been already studied, e.g., stability under overload and overhead of Raft-based synchronization, there are still more critical defects that have been overlooked, e.g., resource leaks and failure contention. In the second part, the modelling abstractions accounting for all the failure modes identified in the qualitative analysis step are provided. Dependability models, in the formalism of SRN, are used to evaluate different dependability metrics, such as steady state availability, failure dynamics, as well as the impact on the user-perceived service availability. Moreover, the applications of proposed SRN models are presented, demonstrating

how the operators and network architects can leverage such models to compare different deployment scenarios and optimize preventive controller software maintenance policies.

5.7.2 Discussion

Threats to validity. The main threat to validity in the presented study is the accuracy of model parameters. While majority of the model parameters are based on the empirical data, and reported values in similar studies, few parameters are based on reasonable assumptions. Hence, the focus is shifted on the methodology and model structure, rather than numerical results. Nevertheless, the quantitative analysis presented in this chapter gives a reasonable estimation of the order of magnitude of the impact of different failure modes. The second threat to validity of the results is the level of abstraction of the proposed models. While the study on Raft response time [161] models the information flow during replication and handover procedures for a single request, the focus of DASON framework is on a wholistic assessment of distributed control plane dependability for a variety of services, replication styles, and on the longer time scale. The chosen level of abstraction represents a compromise between accuracy and generalization power. Modelling of the fine-granularity dependencies, such as relationship between the resource leaks and the serving rate, is left as a part of the future work. Another threat to validity to the presented framework is that the failure manifestation, and consequently the perceived service availability will depend greatly on the use case, e.g. high volume traffic in data center services, traffic engineering in the carrier grade networks or small enterprise networks, will all result in different service mix.

Generalization of the results. The results presented in this chapter can be generalized to many commercial controllers, whose code base contains ONOS and ODL artefacts, such as Ericsson, Huawei and Cisco. Moreover, distributed cores of both controllers, i.e., Atomix [85] and Akka [84], are available as stand-alone open source projects, and could find an application in other distributed systems, beyond network control software. The proposed modelling abstractions for imperfections in distributed SDN control plane and the services running on top of it also have wider application, and can be extended to other distributed systems.

Hopefully, the proposed DASON framework is only the first step towards robust, data-driven, model-based certification of softwarized networks.

Future work. Dependability assurance framework for distributed HA platforms in SDN, presented in this chapter, analyzed defects in the implementation of distributed consensus protocols and control plane recovery procedures. The framework could be extended to support cross-layer optimization, in scenarios where physical network failures cannot be neglected. Another enhancement is a design of sophisticated test-suites that would effectively prevent of the repetitions of the previous errors in implementation of distributed control plane.

Cross-layer optimization of east-west interface in distributed SDN control plane. In the scenarios with dominant network failures, such as unreliable wireless links in WSN or fibre cuts in large geo-distributed Smart Grid systems, the control plane should be designed to be robust to both software and physical failures. Robustness against physical link failures is achieved by installing several disjoint or partially-disjoint paths. In such scenarios, a joint optimization of physical and software fault tolerance

configuration is required, e.g., controller placement, and configuration of Multi Path TCP (MPTCP) [47] and φ-accrual detector [71].

Development of sophisticated testing frameworks for distributed fault tolerance. The significant improvements in failure prevention in distributed systems can be achieved by design of sophisticated test-suits. General test frameworks for distributed environments already exist, such as Jepsen [155] and Chaos Monkey [129]. However, an open research questions is how efficient these tests are in detection of already reported defects against distributed SDN control platforms.

Chapter 6

Software Ageing and Rejuvenation in SDN Orchestration Platforms (ARES)

6.1 Introduction

6.1.1 Motivation, Problem Scope and Research Challenges

The recent trend of network softwarization suggests a radical shift in the implementation of traditional network intelligence. Softwarized network components are expected to provide uninterrupted service during long periods of time, which makes them prone to the effects of *software ageing*, i.e., gradual degradation of the system performance due to an increase of resource consumption, such as memory leaks and unreleased thread locks. The effects of software ageing have been empirically proven in many commercial software solutions, such as modern operating systems, Linux [40] and Android [37], cloud orchestration platforms [25, 172], Apache web server [64, 114, 115], Java Virtual Machine [41, 142] and complex, data-intensive Java-based software, e.g., Hadoop and Cassandra [52, 54].

Ageing related issues have been observed in operational SDN-based networks using ONOS and ODL. Current tests are not efficient in detection of ageing-related performance issues, since they typically require a long time to manifest (order of weeks and months), whereas the continuous integration tests are typically much shorter (order of minutes). Hence, the majority of the ageing-related defects manifest only in the production environment, incurring higher reparation cost and user-perceived system failures. The effects of software ageing can be disastrous for the network; consider the following examples that were reported for SDN controllers:

- *Hard crash* due to resource exhaustion, e.g. out-of-memory error [44][*DesignIssue-1*]
- *Controller becomes unresponsive over time*, caused by unreleased references to device reply messages [*DeadLeak-1*]
- *Gradually degrading performance*, e.g., intent throughput decreasing monotonically in ONOS cluster over time [200][*PerfReg-1*]
- *Cascade of control plane failures*, caused by false positives of time-based failure detectors, which wrongly assume that the slow controller node is unavailable [44, 68]

- *Transient state inconsistencies* between SDN controller replicas and data plane, which can result in loss in data plane performance, e.g., loss of data plane traffic by installing the paths with blackholes [63]
- *DDoS attacks* by enforcing ageing-triggering workloads [*DesignIssue-2*]

Such ageing-related defects have been overlooked in the SoA on dependability of softwarized networks. The main goal of the study presented in this chapter is to provide the first insight into the effects of software ageing in SDN controllers, and evaluate their impact on data plane performance.

6.1.2 Methodology: ARES Framework

This chapter presents ARES, the proposed framework to assess and prevent the effects of ageing in softwarized networks, with the focus on SDN controllers, as illustrated in Fig. 6.1.

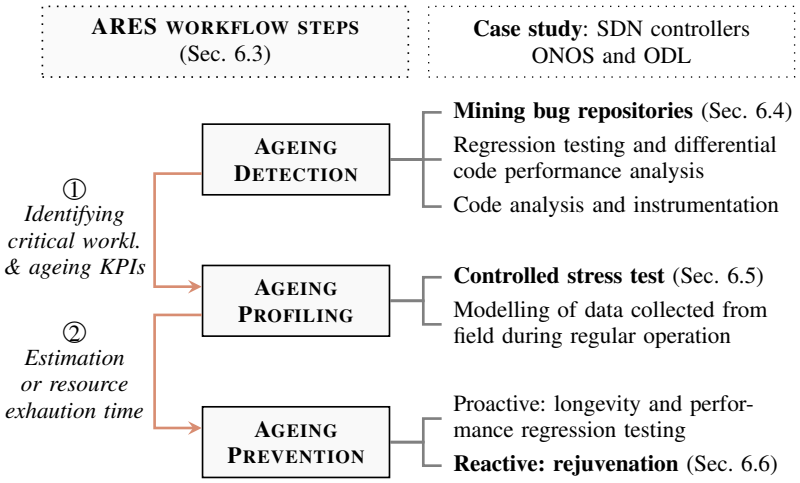


Figure 6.1: ARES: a framework for the management of software ageing consists of three steps: i) detection and localization, ii) profiling the effects of the ageing and iii) prevention. Each step is applied to the particular case study on SDN controllers.

The proposed framework consists of three steps: i) detection of ageing-related issues and identification of critical workload patterns, ii) profiling of software ageing in a measurement-based study and iii) design and implementation of preventive software rejuvenation policies. First, an overview of the ARES framework and the alternatives in each step are discussed in Sec. 6.3. Next, the framework steps have been implemented and evaluated in a case study on open source SDN controllers: ONOS and ODL.

6.1.3 Key Contributions

The contributions of the work presented in this chapter can be summarized as follows:

i. The identification of software ageing patterns and localization of ageing prone software components is performed by mining of code repositories and issue trackers of the two open source

controllers, ONOS and ODL. The analysis has shown that, while many of the ageing related issues stem from bugs (e.g., dead object not being disposed of correctly), two critical ageing related issues are result of deliberate design choices in the implementation of distributed network control plane.

ii. The impact of software ageing on data plane performance is evaluated in a measurement-based study, providing an empirical evidence about non-negligible effects on the network performance and SDN control plane performance and availability.

iii. The design of different preventive rejuvenation policies demonstrates how identified ageing effects can be effectively mitigated in a production environment.

The results presented in this chapter have been submitted to a peer-reviewed journal [4].

The remainder of the chapter is organized as follows. An overview of the related work on software ageing, as well as the limitation of the existing studies on SDN reliability and performance benchmarks are presented in Sec. 6.2. The overview of ARES framework is presented in Sec. 6.3, while individual steps are presented in the following sections: localization of software ageing issues (Sec. 6.4), measurement based study (Sec. 6.5) and implementation of software rejuvenation policies (Sec. 6.6). The chapter concludes with the summary and discussion of the results in Sec. 6.7.

6.2 Related Work

6.2.1 Reliability and Performance Issues in SDN Controllers

Software ageing in reliability modelling of SDN controllers. The majority of studies on reliability in SDN assume either a perfect controller that never fails, or reduce the software to binary states "*operational*" or "*failed*". The reliability of SDN controllers has been addressed only by a limited number of studies [22, 106, 161, 14, 16, 5, 17]. Only the two of them [22, 14] have recognized the problem of software ageing. Vizarreta et al. (author's preliminary work) [14] modelled the impact of software ageing on controller, and demonstrates for which workload intensity it has a non-negligible effect on the controller reliability. Alencar et al. [22] provided the first measurements of the memory consumption and CPU utilization of the research prototype controllers, Floodlight and Beacon, under high load.

Software ageing in measurement-based studies on SDN controllers. The test suites of production grade controllers, Test-ON in ONOS and CSIT in ODL, provide a limited support for detection of ageing defects, which are further discussed in Sec. 6.5. The SDN controller benchmarks tools proposed in SoA, such as CBench [132], OFCbench [87], perfbench[31], MT-CBench [83], and OFCProbe [88, 130] focused on operational KPIs, e.g., maximum throughput or average response time, rather that performance over long hours of operation, which could expose the effects of software ageing.

From a number of studies on performance benchmarks of SDN controllers [23, 27, 44, 68, 92, 93, 164, 167, 168, 170, 177, 200] only the three [44, 168, 200] have reported the issues of software ageing. The first one to encounter the effects of ageing in SDN controller was [168], which showed that while MuL and Maestro started dropping some control plane messages after several minutes,

the other five controllers in the study did not exhibit such performance issues during 24h-test. The testing and benchmarking in realistic telco-cloud setup was elaborated in [44]. The authors tested the performance and scalability of ONOS intent framework, as well as the relationship between the performance and hardware configuration. During one of the experiments, the authors reported an issue of increased heap memory usage (30% of heap used in 5 minutes), under a constant number of active intent requests, but it was not investigated further. The performance degradation under high load led to the instability of control plane in a distributed setup. The slow node was falsely declared as unavailable, due to the time-out based threshold detection[1]. A study by Zhao et al. [200] also reported the unpredictable performance and memory leaks when ONOS and ODL were run over long periods of time. However, the authors do not investigate the issue further and exclude them from performance benchmarks, focusing on simpler controller architectures.

6.2.2 Empirical Studies on Software Ageing

Since the first models of software ageing and rejuvenation by Huang et al. [80] and Garg et al. [50], a multitude of studies on detection and prevention of software ageing effects have emerged. A comprehensive survey of ageing-related studies is presented by Cortaneo et al. [39], while a systematic mapping of the relevant research in the period between 1995-2014 is presented by Valentim et al. [182]. The most relevant empirical studies are summarized in Table 6.1.

Systems under study. The ageing-related issues have been empirically shown to exist in many commercial software solutions: modern operating systems, Linux [40] and Android [37], Unix cluster [181], cloud controllers, both proprietary [172] and open source solutions [25], Apache web server [64, 114, 115] and SOAP server [169], Air-Traffic-Control middleware (CARDAMOM) [32], Java Virtual Machine [41, 142], as well as several large open source `Java` applications, such as Hadoop and Cassandra [53, 54].

Methodology. The methodologies to prove and characterize the effects of software ageing include: i) controlled stress tests, ii) modelling of ageing based on operational measurements, iii) regression testing and differential performance analysis, iii) an empirical analysis of software ageing defects, and iv) code level inspection and instrumentation.

i) Most of the studies used *controlled stress-tests* to statistically prove the effects of ageing, and model the workload-ageing relationship. The main challenge in the feasibility of such studies is how to cover all possible factors (e.g., request types, frequency, environment configuration) that might be encountered in an operational environment, as well as the duration of the experiments required for the ageing effects to manifest. In order to isolate the impact of different factors, and accelerate the ageing by focusing on the most critical ones, the complex Design of Experiments (DoE) plans were proposed. Two phase tests were often used, e.g., first pilot tests were conducted to detect factors to accelerate ageing, as in [37, 114]. In [114] the relationship between resource consumption (e.g., memory) and stress levels (e.g., workload intensity) was used to estimate the Time to (resource) Exhaustion (TTE). The study on the Android operating system [37] covered low- and high-end devices, native, vendor-specific and 3rd party applications and different user-behaviour with only 27 experiments.

[1] A similar problem of stability of distributed control plane under load was reported independently in [68].

Table 6.1: Overview of related work on software ageing.

System	Methodology	Ageing-related KPIs	Remarks
Unix cluster [50, 181]	Data, collected from campus network, is clustered based on system workload descriptors (*sysCall, pageIn/Out, cpuContextSwitch*)	Free memory and used swap space	Workload-ageing relationship modelled as semi-Markov reward process; TTE is obtained by solving a model
Linux OS [40]	Controlled stress test: process-, memory-, file-, networking- and device drivers management	Memory consumption and system call latency	Impact of different workload parameters separated by PCA.
Android OS [37]	Controlled stress test: covering different HW, applications and workload volume and type	Storage and memory usage (system and per process), application launch time	Relevant indicators were per process memory consumption and GC metrics, while storage usage did not show strong correlation
Apache web server [114]	Accelerated stress tests: combination of (http) request rate, page size and type	Workload (stress factors) - ageing (memory depletion) relationship was used to estimate failure times	Achieved acceleration of the tests by factor $\approx \times 100$
IBM Pure cloud platform [172]	Predictive modelling of resource consumption, learned on live data	Memory (swap and real), disk usage, no. of threads and open files, mean response time	Ageing patterns are inferred from bug repositories: storage fragmentation, thread leakage, and degraded response time
Eucalyptus cloud platform [24]	Controlled stress test: management of elastic storage and VM lifecycle	Memory (resident set size, virtual memory, swap space), CPU utilization	Non-negligible memory leak rate was reported
Air-Traffic-Control middleware [32]	Two-phase tests: pilot tests identify most critical factors; exhaustive stress tests to profile ageing for critical workloads	Memory used by system and per process (real and virtual)	Estimated TTE of 4 GB of memory was $\approx$ 2.5 days in the worst case
JVM [41]	Empirical system analysis to localize root cause, controlled experiments to establish workload-ageing relationship	Throughput loss and memory depletion (slow drift due to just-in-time compiler fast drift due to GC)	Estimated TTE for slow depletion was 3 mon.-1 year, for fast depletion 2 days-2 weeks, while throughput halved after a week
5 Java projects [52]	Regression testing, differential analysis for detection and root cause analysis	Memory usage statistics, object allocation and deallocation count	Faster and more robust than trend-based approaches, able to localize root cause, integration with unit tests possible
10 Java projects [54]	Empirical analysis mining public bug repositories (450+ issues)	Memory leaks, mismanagement of Java file handlers, unclosed threads, database and network connections	48% of reported defects are memory-leaks; causes in 80% are non-closed resources, unreleased references and exception handling
3 C/C++ projects [152]	Cross-project transfer learning is used to isolate ageing-prone modules	Training the model on similar projects with larger data set for improved predictive accuracy	Transfer learning showed much better performance that in-project prediction for in all cases

All presented studies are *trend-based*, that is, once the experiment space is designed the ageing trend (e.g., memory depletion) is confirmed using statistical hypothesis testing, e.g., Mann-Kendall and its variations. If a trend is detected, it is quantified using statistical inference techniques. Typical models used in such analysis are linear, quadratic, piece-wise linear, growth curve, ARIMA, queuing models, fractal theory. Model parameters were fit using LSE/MLE or more robust Sen's slope estimator.

ii) The studies modelling the ageing on data collected from the operational environment *"in the wild"* deal with more representative workload. The study on IBM cloud platform collected the data from multiple customer premises [172], while the UNIX study [181] gathers data from several campus machines. Data points are then clustered around system load parameters (e.g., *cpuLoad*) to describe representative operational points.

iii) *Regression testing and differential performance analysis* is an alternative approach for detection and localization of ageing-related problems [52, 101, 102, 113]. Ageing is detected by comparing memory usage distribution, byte level allocations, signal divergence charts between baseline and buggy version of the software. The differential analysis in [101] was performed comparing runtime results of memory consumption in fault-free and buggy versions of cloud platform, while the study in [102] used fault injection to demonstrate the efficiency of differential unit tests, by monitoring the heap allocation. A similar approach was validated on 7 real-life leak-bugs reported in 5 Java projects [52]. Furthermore, the study in [113] used signal divergence charts for anomaly detection, i.e., a critical level of deviation between target and baseline version.

iv) *An empirical analysis of software ageing defects* are conducted with a goal to analyse the patterns, improve testing tools, and prevent occurrence of the same issues in future releases. The largest empirical study on ageing in 10 open source Java projects was presented in [54] (ext. of [53]). The patterns in ageing manifestation, root causes and repair strategies were observed by manual inspection of 450+ bugs. An empirical analysis using cross-project transfer learning was proposed in [152], with the goal of isolation of ageing-prone modules. Another example of cross-project learning was [38] using the software complexity to predict ageing-related defects.

v) The effects of software ageing manifesting as memory-leaks can be detected with *code level inspection and instrumentation*. LeakBot [124] and LeakChaser [191] were proposed for automated leak-detection, while memory-leak detection based static code analysis with LeakChecker was discussed in [196]. This approach can efficiently localize the root causes much faster than trend-based tests in controlled environment.

6.3 ARES: A Framework for Management of Software Ageing and Rejuvenation

The effects of software ageing have been analysed and experimentally proven in well-known applications, similar to SDN controllers in terms of complexity (i.e., code base size), functional characteristics (e.g., distributed stores and operating systems), and usage patterns (e.g., data intensive long running systems). The ARES framework presents a systematic way for the management of software ageing, by combining the three steps, i.e., ageing detection, profiling and prevention, which have not been

previously applied together in the SoA. The alternative approaches in each step (illustrated in Fig. 6.1) are discussed, comparing their advantages and limitations.

6.3.1 Detection of Software Ageing

A timely detection of software ageing defects is challenging. The ageing defects take long time to manifest, as they represent the cumulative effect of comparatively small resource leaks, which are sometimes triggered only after a particular sequence of events, e.g., an unsuccessful object removal under exceptions. The methods for detection and localizations of ageing related defects are: i) empirical analysis of previous defects, ii) regression testing and differential code analysis, and ii) code inspection and instrumentation.

The bug repositories contain large corpus of data on the incidents from test and operational environments, providing a valuable insight into manifestation patterns, root causes and reparation strategies [54]. In the era of network softwarization many of the commercial network components at all layers of the networking are based on open source solutions[2]. These open source network projects maintain public issue trackers, reporting the software failures detected in both test and operational phase. Typical ageing-related issues and the critical workloads are identified by mining such repositories. The keyword based search was used to filter the potential software ageing issues from large bug repositories. After the filtering, the manual inspection is employed to identify the real ageing defects. It is left for the future work to automate this step by using NLP.

The regression based tests are rendered inefficient for large changes in code base, due to new features and changes in data structures (e.g., a change in ODL data store abstractions `AD-SAL` → `MD-SAL`). The methods for fault localization based on code instrumentation induce performance issues, require sophisticated statistical analysis, involvement of the developer or are proprietary (e.g., LeakBot IBM). Moreover, the regression testing and static code analysis are not likely to detect the bugs that are triggered only after a specific sequence of events. On the other hand, the main drawbacks that of the empirical analysis is that it only addresses known issues, not being able to anticipate new classes of bugs. Hence, it is left for the future work to explore the efficiency of alternative approaches for localization of software ageing issues.

6.3.2 Profiling of Software Ageing

Ageing profiling is done through an exhaustive measurement campaign, modelling the relationship between workload parameters and the effects of software ageing, as illustrated in Fig. 6.2. The methodology for characterization of software ageing proposed in the literature uses either: i) stress based tests in a controlled environment, or ii) modelling of data collected from an operational environment.

In controlled stress tests the performance and resource consumption under constant workload are monitored for a long period of time. The majority of such tests uses sophisticated statistical methods to detect trends, e.g., memory consumption to detect memory leaks. The main drawback of trend-based ageing profiling is duration of such tests, especially as large number of repetitions are needed to obtain statistically relevant data for different operational points. Moreover, the increased resource

[2]The Linux Foundation: Open Source Networking

consumption does not necessarily stems from ageing, e.g., data structures like cache can keep memory for legitimate reasons [52]. An alternative approach is modelling of data collected from an operational environment. However, the network operators are not willing to provide an access or publish sensitive data, as Google's B4 [63].

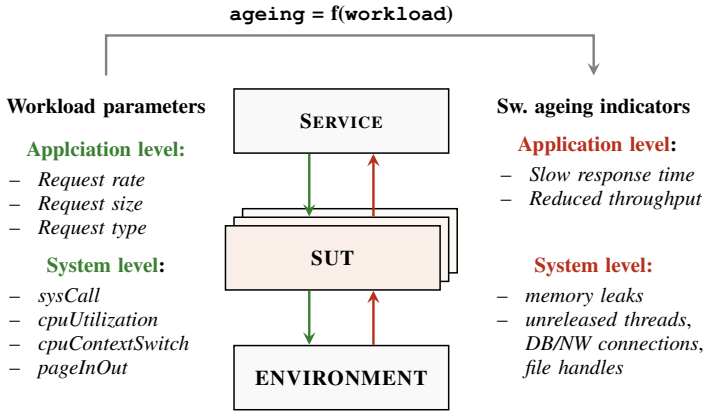


Figure 6.2: Profiling: modelling the workload-ageing relationship for *Software Under Test (SUT)*

6.3.3 Prevention of Software Ageing

The prevention and mitigation of the adverse effects of software ageing can be done either proactively, before the software is released to the customers, or reactively, after the software has been deployed in an operational environment.

The detection of ageing-related problems with standard performance unit/integration test has been shown to be unsuccessful. The ageing effects take long time to manifest as significant performance degradation, while most systems using continuous integration require much shorter time for the tests to pass. Dedicated tests, such as longevity tests used in production SDN platforms, such as ONOS and ODL, test the system for several hours at the time. However, existing tests are focused mainly on stability, i.e., whether a system is working correctly, rather then on effects of the software ageing. The longevity tests in ONOS consists of several steps; one of them is occasional crash of the controller nodes, aiming to assess the control plane stability and correctness, while the system recovers. However, this test sequence reinitializes controller, and unintentionally prevents the manifestation of software ageing effects, such as memory leaks.

The reactive approach implements rejuvenation policies, i.e., occasionally stopping the software, reinitializing internal data structures, network connections and/or restarting the system. The design of software rejuvenation policies consists in optimization of *rejuvenation schedule* and *rejuvenation level*, as illustrated in Fig. 6.3. The schedule refers to the rejuvenation timing, and can be time- or threshold-based. Time-based rejuvenation can be triggered periodically in regular time intervals, after a certain number of operations have been executed (e.g., network intent compilation and installation), or at the times when it has the least impact on the services. Threshold-based rejuvenation is triggered before the resource utilization (real or projected one) reaches the critical level. The rejuvenation can be

implemented in different levels, *application level* (e.g., purging of the data stores) or *system level* (e.g., VM restart). Application level restarts helps mitigation of a particular ageing effect (e.g., memory leak in the data store), while system level rejuvenation helps in prevention of most ageing effects, both known and unknown. The system level rejuvenation should be avoided, as it leads to significantly longer service interruption times.

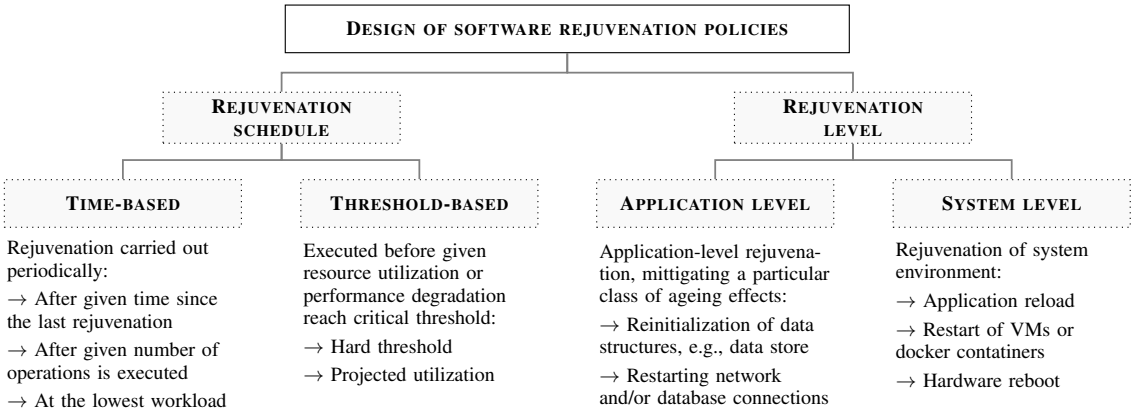


Figure 6.3: Designing of software rejuvenation policies, i.e., optimization of rejuvenation schedule and level.

The following sections discuss implementation of the framework in SDN controllers' case study.

6.4 Ageing Detection: Mining ONOS and ODL Software Repositories

The network components in this study are open source SDN controllers, ONOS and ODL, whose bug and code repositories are publicly available. The bug repositories contain data on the reported issues in test and operational environments, which makes it the primary source for discovery of the ageing patterns. The public code repositories have been used to analyze and explain the nature of ageing root causes, in particular, in cases when the description in the bug repositories were not sufficient.

A large number of the ageing related issues are caught already in the testing phase. Both controllers implement test suites to detect the performance regression and resource leaks as a part of CI process. ONOS longevity tests, called Continuous Hours of Operation (CHO) tests, are implemented as a part of its testing framework, called TestON. The CHO tests use `ChaosMonkey`, which exercises random control plane inputs, such as adding and removal of the flows, topology events (e.g., port, link or device down), and cluster instance down, aimed to run several weeks at the time. Its counterpart, ODL recommends the implementation of longevity tests at the individual project level. However, these tests focus mainly on stability and correctness, and therefore overlook some of critical ageing-related issues, which still manifest in production environment.

Performance studies on ODL[3] and ONOS[4], as well as several measurement-based studies [44, 168, 200] have already reported effects of software ageing in the operational setting, but did not investigate these issues further.

[3]OpenDaylight: How Performance Testing Improved the Nitrogen Release (October 24, 2017).
[4]ONOS: Security and Performance Analysis (September 19, 2017).

The following section describes the methodology to systematically analyze the nature and manifestation patterns of software ageing in such open source SDN controllers.

6.4.1 Methodology for Mining of the Software Repositories

Public issue trackers contain more than 10k+ bugs reported in two repositories over five years (2013-2019). Since the amount of data in these public repositories is rather large, the manual inspection is infeasible. Hence, the filtering of the ageing related issues was implemented using the keyword based search presented in Fig. 6.4. Note, that some keywords are use case specific, e.g., ONOS CHO tests or `Java` garbage collection[5].

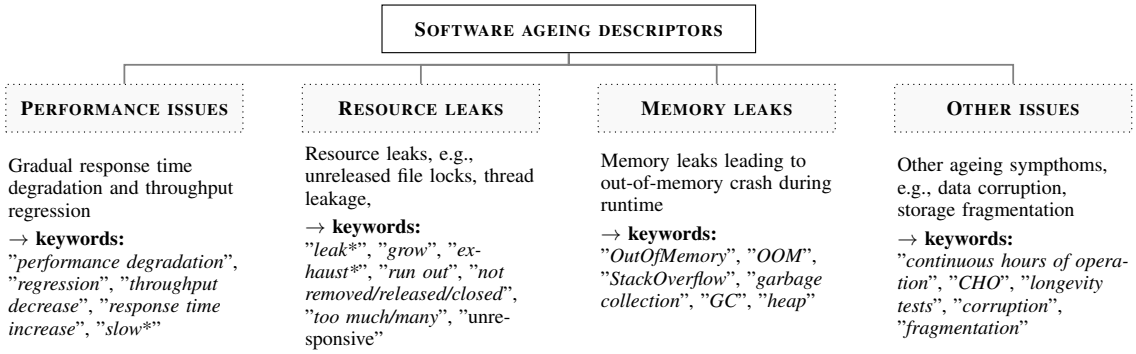


Figure 6.4: Mining software repositories for ageing defects using keyword-based search in filtering step.

After the keyword-based filtering step, the bug repository was reduced to 500 bugs representing the potential software ageing defects. In the following step, the manual inspection was deployed to identify the real ageing defects.

6.4.2 Analysis of Ageing-related Defects

This section provides the first insight into ageing dynamics and recurring patterns of critical issues in SDN controllers. The most representative examples in each one of the categories illustrated in Fig. 6.4 are discussed in the following section.

6.4.2.1 Performance Issues

Gradual performance degradation, reflected in *throughput decrease* or *response time increase*, is the most obvious consequence of software ageing, perceived directly by the user. An example is [*PerfReg-1*], reporting that the overall intent throughput starts decreasing monotonically 5 minutes after the start of the throughput test in cluster mode. The issue does not happen when only single instance is deployed. However, such reports are rare, as most of the ageing-related defects have been reported in the form of resource leaks.

[5] A comprehensive taxonomy of ageing-related defects was provided by [54], and was partially used to design a keyword search space used in the filtering step.

6.4.2.2 Resource Leaks (Other than Memory)

A typical example of the storage leak is reported in ONOS [*Storage-1*]. The controller nodes could eventually run out of disk space, due to a bug in the log compaction. The log compaction removes entries which do not contribute to current state (see Sec. 5.3), and the bug prevented compaction from being rescheduled after completion. In another example [*Storage-2*], the controller node becomes unresponsive with `StorageException`, when the connected network device state changes repeatedly. This was observed only after running the test for several days.

Some leaks happen only under exceptional conditions. E.g., in [*FileHandle-1*] it was observed that some errors during start the `NETCONF` session, cause `SshClient` not to free the sockets or file descriptors, that were made by that client. It led to the exhaustion of file descriptors, preventing the creation of new network connections. A similar issue was reported in [*Session-1*]. In case of the `NETCONF` password error, the device would not be connected. The attempted device registration creates new `ProxySession` without releasing the earlier one.

Leaks introduced after an upgrade can be detected with performance regression tests. In [*Session-2*] a data store was opening a new session on each transaction, after upgrade of ONOS distributed core (Atomix).

6.4.2.3 Memory Leaks

Memory leaks represent the largest category of the ageing-related defects, and are hence, analyzed separately from other resource leaks.

Datastore transaction leaks in ODL

A large number of memory leak issues related to unclosed transactions to data store was reported in ODL. An umbrella issue [*TxLeak-1*] groups all suspected transaction leaks, in an effort to identify the root cause and mitigation techniques. A detailed analysis of transaction leaks and a corresponding debugging tool to trace their source, was provided by RedHat[6]. Their analysis showed that embedded applications, such `OpenFlowPlugin`, open a large number of transactions, but never close them, filling up an internal data structure, and leading to the crash with an `OutOfMemoryError (OOM)`.

A failure of the existing test frameworks to detect ageing-related bugs was discussed in [*TxLeak-2*], which reported an `OOM` due to a large hash map in `ShardDataTree`, suspected that the `DataBroker` was not closing the transactions correctly. It was speculated that CSIT longevity test either does not run long enough, or does not stress the data store at the critical levels.

An `OOM` [*TxLeak-3*] caused by a large number of closed transactions was reported in history's of closed transactions. The transactions were created by the `ask-based` (pull) protocol, and were closed, but not purged from the hash map. On another hand, `tell-based` (push) protocol issues explicit purges from the frontend, while for `ask-based` protocol this has to be done on the backend (for functional split between frontend and backend in ODL data store, see Fig.6.5).

Transaction leaks do not always result in hard failures due to `OOM`. Silent failures, such as [*TxLeak-3*] are much more dangerous, as they can go unnoticed by standard failure detectors. During some longevity tests in a clustered ODL setup, the controllers were up, with ports listening but not being functional.

[6]Michael Vorburger (RedHat) How to find Transaction related memory leaks in OpenDaylight?

Dead objects not being disposed of correctly

A large empirical study on ageing-related defects in `Java`-based applications [53, 54] showed that dead objects not being disposed of correctly, contribute to 20% ageing bugs. Object disposal may be unsuccessful for several reasons: due to a omission failure, a bug (e.g., unreleased reference), or under exceptional conditions.

A typical example is [*DeadLeak-1*], which reports the `NETCONF` protocol handler not releasing the references to replies, leading to memory leaks which caused ONOS to become unresponsive over time. In some cases, the failed removals happen intermittently, as in [*DeadLeak-2*]. This report describes a failure of intent removal in the CHO test, when a large number of the network intents are installed. Reproducing this issue was hard, as it manifested in "less than 1/10000" cases, and it remained unresolved.

Sometimes, the ageing effects are observed in functional tests. A test case called "*chasing the leader*", aiming to establish whether a service operates correctly when faced with rapid mastership transitions, uncovered several ageing-related issues. For instance, [*DeadLeak-3*] reported that *chasing the leader* exhausts heap space in 19 hours. Performance issues, such as "karaf.log files show UnreachableMember", corresponding to `GC` pauses of more than five seconds, started happening earlier, 3.5 hours after the test start.

Failed object removals under exceptional conditions are extremely hard to reproduce in the test environment, because their triggering require the precise timing between the internal and external events in the control plane. Memory leak in `DistributedMeterStore` was reported in [*ExceptionLeak-1*], observing that the *CompletableFuture* (i.e., asynchronous calls) were not removed when the futures were completed exceptionally. Another example is [*ExceptionLeak-2*], which reports the controller failing to remove flows from `DistributedFlowRuleStore` for disconnected device, and keeps re-adding the flows when device reconnects. Similarly, [*ExceptionLeak-3*] reports a failure to remove flows installed in multiple times in several instances.

Ageing-related issues induced by the distributed core design

All presented ageing-related issues until this point are clear bugs, an undesired behaviour of the controller software that should be corrected. However, the analysis exposed two highly critical ageing-related issues that are caused by the particular design of distributed core in ONOS and ODL. These two issues lead to a deliberate increase in the memory footprint of data stores, which degrades the controller response time, and eventually leads to a crash with an `OOM`.

The design issue in ONOS [*DesignIssue-1*] was introduced to ensure the stability of distributed primitives using `EventuallyConsistentMap`, which rely on Gossip protocol for synchronization (as described in Sec. 5.3). The entries removed from such maps are replaced with *tombstones*, i.e., placeholders, to ensure that the entries do not reappear later when anti-entropy runs (i.e., random pairs of neighbours comparing their version of the data). Adding and removing a large number of unique entries, e.g., flows, lead to an accumulation of tombstones, resulting in memory leaks. The mapping of the network state to ONOS distributed primitives is presented in Table 6.2.

The design issue in ODL [*DesignIssue-2*] stems from the functional role distribution between its data stores, illustrated in Fig. 6.5. Essentially, there are two conceptually different data stores, the `ConfigurationalDS`, which contains the desired network configuration, as specified by the user, and

Table 6.2: The use of distributed primitives in ONOS (adapted from "ONOS Distributed Core" presentation by Thomas Vachsuka)

Network state	Consistency level	Properties
Network topology	*Eventually consistent*	low latency access
Flow rules, flow statistics	*Eventually consistent*	shardable, soft state
Switch-controller mapping	*Strongly consistent*	slow changing
Application intents	*Strongly consistent*	durable, immutable

the `OperationalDS`, which contains the actual state of the network. A user can request the installation of flow rules with hard and idle timeouts, which would appear in both data stores after a successful installation. When the flows "expire", i.e., timeout, they are removed from the `OperationalDS` (and the network), but they remain in the `ConfigurationalDS`. The issue was identified as a potential security vulnerability, making the controller prone to overflow attacks. The controller overflow attacks have much larger impact than DDoS attacks in data plane, e.g., flow-table-flooding attack. The report contains an elaborated discussion about the design trade-off(s), arguing that the rate limiters for the addition of the "expiring" flows should be implemented outside the data stores, or preferably outside the controller.

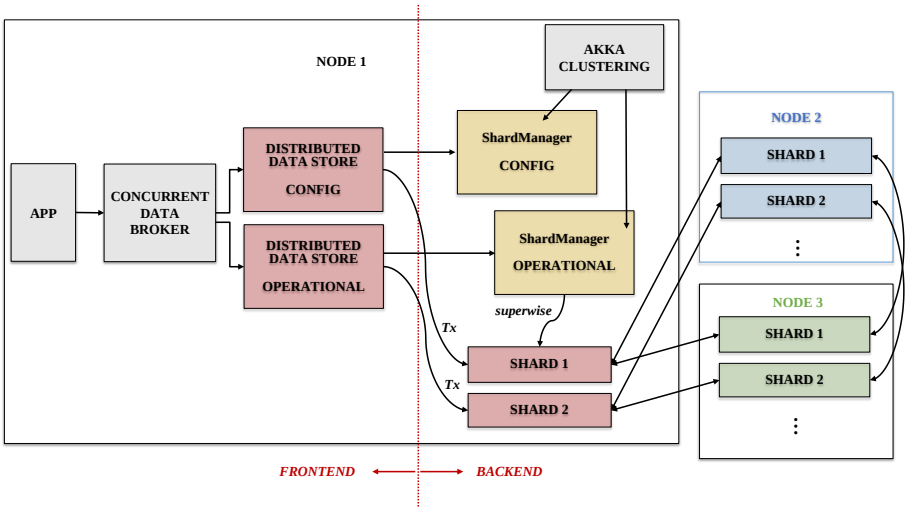


Figure 6.5: ODL data store architecture (adapted from "Clustering in OpenDaylight" presentation by Jan Medved and Robert Varga).

Unlike the previous issues that were rectified, ageing-related design issues (ONOS [*DesignIssue-1*] and ODL [*DesignIssue-2*]) are present in operational controller software, and will inevitably have an effect on the network performance. The goal of the following section is to evaluate whether these issues have a non-negligible impact on the network performance, referred to in this chapter as *network ageing*.

6.5 Measurement-based Characterization of Network Ageing

This section presents the measurement-based characterization of the effects of network ageing. The focus of the study are the ageing defects stemming from the design issues in distributed control plane implementations of ONOS and ODL. First, a design of experiments is presented (Sec. 6.5.1), followed by an overview of the testbed implementation (Sec. 6.5.2). The evaluation of the network ageing effects for the critical workloads is finally presented in Sec. 6.5.3.

6.5.1 Design of Experiments (DoE)

DoE, or experimental design, aims to uncover the relationship between relevant *factors* (e.g., request type: intent or expiring flow), stress *levels* (e.g., batch size or arrival rate) and measurable *response* (e.g., memory depletion and response time).

The critical workloads identified in the previous section, stemming from two design issues in distributed control plane implementations of ONOS and ODL, aim to enforce the memory leaks in the controllers' data stores. The corresponding DoE design and implementations are elaborated in the following sections.

6.5.1.1 Experiment Design for *DesignIssue-1* (ONOS-4212)

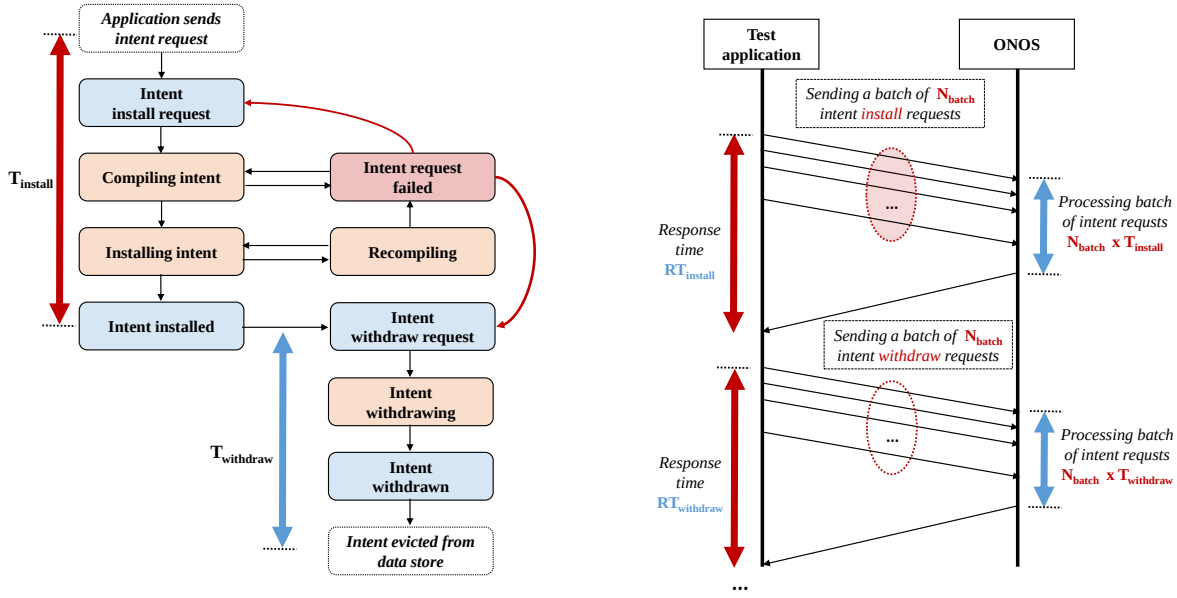
Adding and removing unique flow entries in ONOS, trigger the accumulation of the tombstones, instead of fully deleting the flow entries, ensuring the stability of Gossip protocol. In order to further stress the memory depletion and observe the controller response time, the intents, i.e., a higher layer flow abstraction in ONOS[7], are used instead. The experiment design and response time measurement of ONOS intent installation/withdrawal times are illustrated in Fig. 6.6.

Measurement points for the intent installation ($T_{install}$) and withdrawal ($T_{withdraw}$) times are illustrated in Fig. 6.6a. After the intent installation is requested by an application, it is processed and compiled to the low level flow rules. In case of unsuccessful compilation, the intent can be recompiled several times, and eventually fail. The application can also request a removal of the intent, after which it is withdrawn from the network and evicted from data store.

The interaction between the test application, requesting the installation and withdrawal, is illustrated in Fig. 6.6b. A test application requests a batch of N_{batch} intents to be installed, waits for ONOS to install the intents, and subsequently requests a removal of the batch of the intents. The observed response time ($RT_{install}$) includes not only the real processing time (e.g., $N_{batch} \times T_{install}$), but also the system overhead and the propagation delay between test application and ONOS, which reside on different machines (see Sec. 6.5.2). Hence, the batch size has to be carefully tuned, in order to analyze the trend in intent installation and withdrawal response times.

The ONOS built-in test application `push-test-intents`, which is a part of TestON plan, was modified to keep sending the intent installation and withdrawal request, omitting the warm-up phase and restarts between batch requests. The reference topology consists of seven stub switches (called `Null Providers`). Different classes of intents, e.g., point-to-point, host-to-host, are implemented in

[7] ONOS Intent Framework: allows applications to specify higher layer policies, in terms of resources, constraints and instructions, which are translated to lower level flow rules via intent compilation.



(a) Response time per intent

(b) Response time per batch of intents

Figure 6.6: Measurement points for response time evaluation for ONOS intent installation ($T_{install}$) and withdrawal ($T_{withdraw}$) times. Intent state transition diagram was adapter from ONOS documentation.

ONOS. The experiment considered a batch of point-to-point intents installed between the first and the last switch in the linear topology. The experiments are run in a throughput mode, i.e., a new batch of requests is sent as soon as the old batch is processed. The stress level can be adjusted by changing the intent batch size.

6.5.1.2 Experiment Design for *DesignIssue-2* (OPNFLWPLUG-962)

The second set of experiments tests whether the installation of the flows with hard timeout triggers the memory growth of the `ConfigurationalDS` in ODL. The overview of the experiment design is illustrated in Fig. 6.7.

The test application sends a batch of N_{batch} expiring flows with hard timeout $T_{timeout}$. The flows are recorded in the `ConfigurationalDS`, which is the only data store that the user can modify. After the flows have been pushed and installed in the network, they are added to the `OperationalDS` and their state is updated. After the hard timeout $T_{timeout}$, the flows are removed from the network and the `OperationalDS`, and their state is updated. However, the `ConfigurationalDS` retains the expired flows, unless the user or the application explicitly request their removal.

The requests are sent in regular time internals $T_{wait} > T_{timeout}$, ensuring that the flows have expired, before adding a new batch and causing an overload. The observable response KPIs is the memory consumption, whose growth is expected due to the accumulation of the expired flows in the `ConfigurationalDS`. The stress level is controlled by changing the flow batch size and flow arrival rate ($1/T_{wait}$). The data plane was emulated using Mininet[8].

[8]Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native): http://mininet.org/

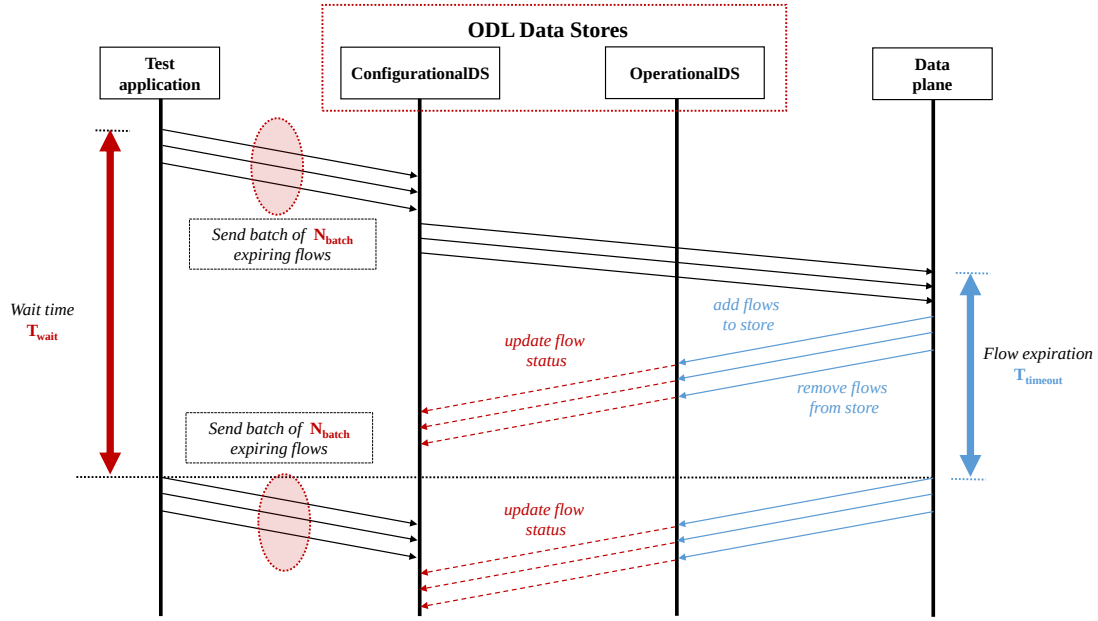


Figure 6.7: Experiment design for ODL issue: a batch of N_{batch} expiring flows with hard timeout $T_{timeout}$ is added to the network in regular time internals T_{wait}.

Since the addition of the expiring flows with the hard timeout at the constant rate is expected to keep the controller memory consumption constant, it is convenient for testing of the presence of memory leaks. Hence, the similar experiment was exercised for ONOS controller. The hypothesis that expired flows would lead to the increase in memory consumption of the FlowStore, due to the similar mechanism as in *DesignIssue-1*.

Another variable factor in both experiments is the controller software version. The exhaustive analysis could not be conducted for all software releases, due to the duration of the experiments. For instance, a 24-hour experiment with 10 repetitions for the last six controller versions would take two months to complete.

6.5.2 Testbed Setup and Implementation

The testbed consists of three workstations, as illustrated in Fig. 6.8. **Management PC** sets up the environment (docker images of controllers under test, workload generator, and tools for stats collection and logging). **Workload PC** emulates SDN Control Plane (CP) traffic using different benchmark tools and scripts. **SUT PC** contains the controller under test, i.e., different versions of ONOS and ODL. Experiment statistics and logs are collected using *telegraph*[9]/*influxDB*[10] and *fluentd*[11], and visualised with *Grafana*[12].

The Management and the Workload PCs have Intel(R) i5-3470 CPU @3.40GHz processors, while the Controller PC has i7-3770 CPU @3.40GHz. The isolation between the applications is ensured

[9] Telegraf is the agent for collecting and reporting metrics and data
[10] Influx DB is purpose-built time-series DB
[11] fluentd is an open source data collector for unified logging layer
[12] Grafana is an open platform for analytics and monitoring

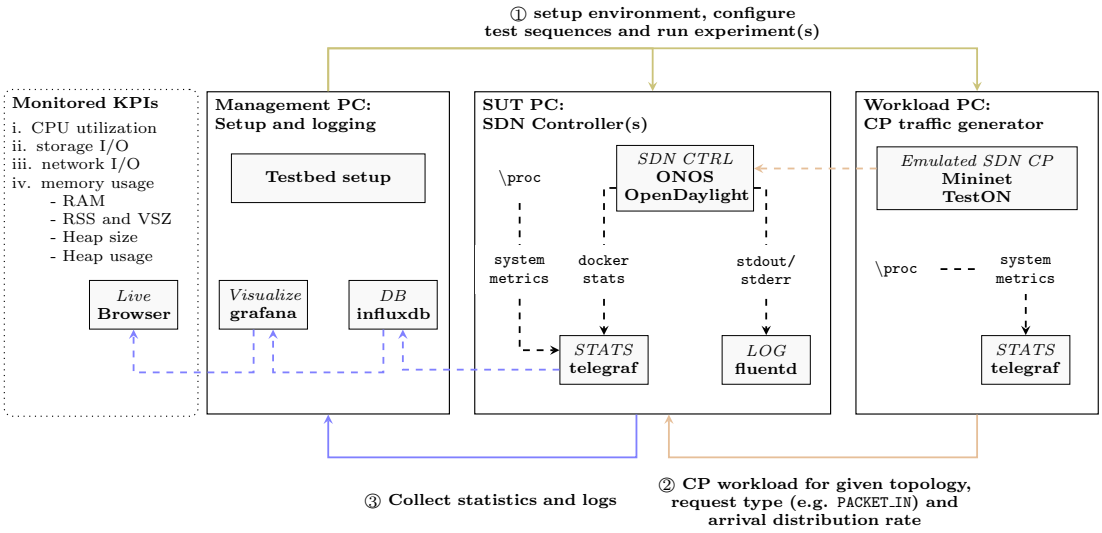


Figure 6.8: Testbed consists of three workstations: *i) management PC* responsible for environment setup, configuration of test sequences and post-experiment stats collection, *ii) workload PC* generating SDN CP traffic and *iii) SUT PC* running the containers for SDN controller under test and statistics collection.

by dedicated core mapping. Within SUT PC, the container with the controller under test is using cores 1-7, while *telegraf* and *fluentd* are mapped to the core 0. All workstations have 16 GB Random Access Memory (RAM) and use Ubuntu-18.04 operating system; a desktop version is installed on the Management PC, and server on other two machines.

During the experiments, the logs and the metrics on the system and application levels are collected. The KPIs collected from SUT PC (CPU usage, number of threads and open files, disk utilization and most importantly used memory) and workload PC (response time), are reported to the management PC. Since the focus of the presented study is on the memory consumption, due to the nature of the ageing-related defects, memory-related KPIs are monitored with higher granularity. A brief comparison between memory-related KPIs is provided below:

1. RAM: the main system memory is too noisy to monitor the memory leak trends, as SUT process(es) only use a fraction of it.
2. Resident Set Size (RSS): the working set size of a monitored process, considering text, data, stack and heap loaded in main memory.
3. Virtual Memory Size (VSZ): VSZ represents the total amount of the memory, including in-memory and on-disk pages.
4. Heap Size (HSZ) and Heap Usage (HUS) total amount of allocated and used heap, are less noisy than the previous three metrics [113].

The system wide metrics are collected from the operating system and docker containers, while the details of memory consumption are obtained through Garbage Collection (GC) logs[13]. The workstations are setup to use Concurrent Mark and Sweep (CMS) Incremental Mode. The maximum heap size was 14GB, which was the amount of allocated RAM for a docker container running the controller.

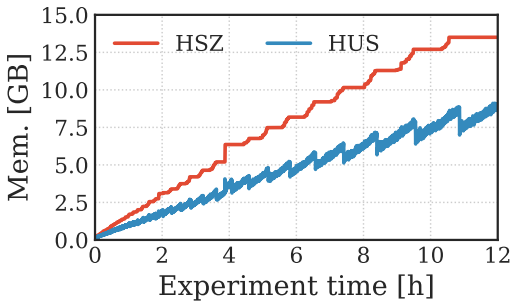
[13]For details see: What Is a Garbage Collection Log and How Can You Enable and Analyze It?

6.5.3 Characterization of Software Ageing

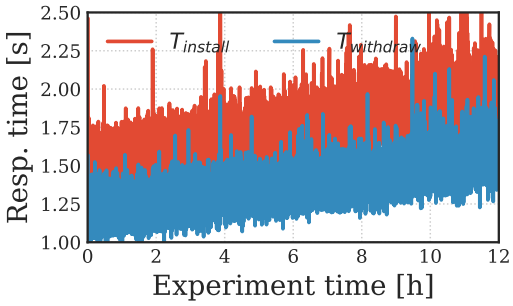
Next, the impact of the controller software ageing issues on the network performance is presented.

6.5.3.1 Evaluation of ONOS Design Issue

The design issue in ONOS (*DesignIssue-1*) stems from a deliberate choice to replace dead objects in `ECMap`, for the sake of stability of Gossip protocol. Addition and removal of the unique flows or intents lead to the growth of the memory consumption, and consequently response time degradation, as illustrated in Fig. 6.9. The results in the figure are reported for ONOS v1.10.4 (Kingfisher) and batch size of $N_{batch} = 1000$ intents.



(a) Memory depletion: allocated heap size (HSZ) and heap usage (HUS)



(b) Response time degradation: installation and withdrawal of intents

Figure 6.9: Network ageing in case of *DesignIssue-1* (ONOS-4212)

The memory consumption over a 12h experiment time is illustrated in Fig. 6.9a. The memory consumption piecewise linear behaviour and a long term growth trend can be observed in all reported memory KPIs. The memory depletion rate in this experiment is $0.78\frac{GB}{h}$, which is equivalent to $18\frac{GB}{day}$, at the effective rate of $300\frac{intents}{s}$.

The response time in terms of *intent installation* and *intent withdrawal rate* is presented in Fig. 6.9b. Similarly, a long term growth trend can be observed. The response time degradation time is non-negligible, estimated to increase of 50%/day for the intent installation time after 24 h of operation.

In order to confirm whether the estimated memory depletion trend is stable and it does not saturate under high memory utilization, the experiment was left to run until the Out Of Memory (OOM) crash. The memory depletion trend remained linear until the crash, which happened after the predicted 18 h. However, the response time increase slowed down around 14 h after the experiment start.

6.5.3.2 Evaluation of ODL Design Issue

The design issue in ODL (*DesignIssue-2*) has a provenance to keep a desired flow configuration provided in `ConfigurationalDS`, regardless of the operational flow state. This causes a constant growth in the memory consumption in case of addition of expiring flow rules, i.e., the flow rules with hard or idle timeout. The effects of this issue on the memory consumption are illustrated in Fig. 6.10. The results reported in the figure are for ODL v0.7 (Nitrogen), the flow batch size of $N_{batch} = 1000$ and experiment duration of 12 h. The effective flow arrival rate was 100 $\frac{flows}{s}$.

It can be observed that all memory KPIs show piecewise linear behaviour, with a long term growth trend. The observed memory depletion rate in this experiment was 18 $\frac{\text{GB}}{\text{day}}$, similar to the previous case.

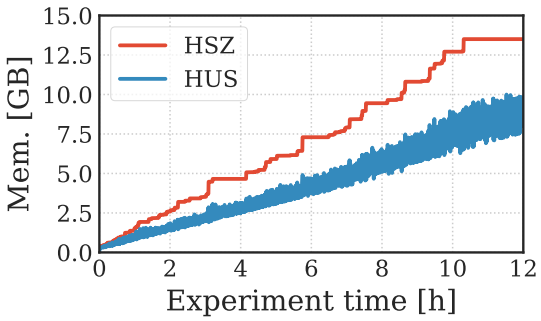


Figure 6.10: Network ageing in case of *DesignIssue-2* (OPNFLWPLUG-962)

Note, that the underlying root causes of software ageing in case of expiring flows in ONOS and ODL are different. Both experiments can be easily integrated into ONOS and ODL test suites. In particular, while the presented design issues remain in the controller software, the network operators would benefit from knowing the expected memory leak rates, as well as the response degradation, and design their systems and preventive measures accordingly.

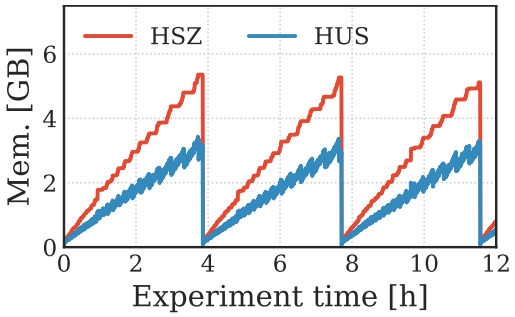
The exhaustive search of the software ageing in different operational points is left for future work.

6.6 Design of Rejuvenation Policies

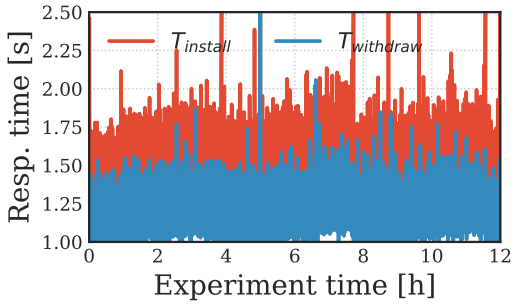
Next, the proof-of-concept implementation and the trade-off between efficiency and service interruption time for different rejuvenation policies are presented.

6.6.1 Proof-of-Concept Implementation

As a proof of concept, a simple threshold-based policy is implemented, triggering the system level rejuvenation after a memory reaches a certain level of utilization. A restart of the docker container, in which the controller is running, is triggered after the RSS reaches 4 GB. The efficiency of this rejuvenation policy in case of ONOS *DesignIssue-1* is illustrated in Fig. 6.11a and Fig. 6.11b.



(a) Memory depletion: allocated heap size (HSZ) and heap usage (HUS)



(b) Response time degradation: installation and withdrawal of intents

Figure 6.11: Network rejuvenation in case of *DesignIssue-1* (ONOS-4212)

In this setting, the rejuvenation was triggered every 4 h, leading to a service interruption times of 30 seconds. Note that in the examples of ONOS (Fig. 6.9) and ODL (Fig. 6.10) design issues, the estimated TTE= 18 h, which is the longest period that the software rejuvenation should be postponed, leading to a controller service unavailability of 0.05%. During these short interruption times, the network is left without control plane services, which can be mitigated by a simple handover to an identical (albeit younger) controller replica. Design and implementation of the protocol for a seamless handover of differently aged controller replicas is left for the future work.

6.6.2 Discussion: Rejuvenation Policy Design Trade-off

Next, the trade-off between efficiency and service interruption time for different rejuvenation policies (Fig. 6.3), in a given scenario are compared.

Rejuvenation schedule. The rejuvenation schedules differ in the triggering event, which has an impact on the implementation complexity and the intrusiveness of the monitoring processes.

In case of time-based policies, the rejuvenation timing is an important design parameter, which is highly dependent on the system workload. The time-based rejuvenation executed in regular intervals, is the easiest to implement, as it can be triggered externally, without the need to monitor the internal system state. In the examples of ONOS and ODL design issues, presented in Fig. 6.9 and Fig. 6.10, the respective memory leak rate is constant, as long as the arrival of critical (i.e., ageing triggering) requests is constant. Is such scenario simple time-based rejuvenation would suffice. However, such periodic rejuvenation is efficient only in the case of the constant ageing rates, which holds only for a Machine-To-Machine (M2M) communication, such as in industrial networks. In systems with variable workload, and consequently variable ageing rate, the inspection-based rejuvenation policy must be implemented instead.

Rejuvenation level. The rejuvenation levels differ in terms of service interruption times, as well as their efficiency to mitigate different types of ageing, e.g., storage v.s. memory leak.

The application level rejuvenation in case of the presented SDN controllers can be done at the data store level (e.g., explicit purging of the expired flows), the application bundle level (e.g., `OpenFlowPlugin`) or at the controller level. The rejuvenation of the data store level is the fastest, leading to negligible service interruption times, but can only deal with the effect of memory leaks, such as *DesignIssue-1* and *DesignIssue-2*. Rejuvenation at the application bundle level can additionally mitigate the effect of resource leaks caused by the misbehaving controller applications, such as those reported in [*FileHandle-1*, *Session-1*, *TxLeak-1*]. The complete controller restart is even more efficient, but leads to significantly longer interruption times [161]. System level rejuvenation can additionally mitigate the ageing effects, such as storage leaks and fragmentation [*Storage-1*]. The service interruption time is an order of magnitude longer than the application level restart, resulting in the outages of several minutes [127].

Hybrid rejuvenation policies. In operational environments, where the exact network ageing rates and its manifestation patterns are unknown, the optimal strategy would be to implement hybrid rejuvenation policies. For instance, short data store clean-ups can be performed frequently to mitigate the known effects of data store leaks, the controller can be restarted periodically preventing the other resource leaks, while the entire system can be reloaded at the times of low workload periods.

6.7 Concluding Remarks

6.7.1 Summary

This chapter presents ARES, the first framework for management of software ageing and rejuvenation for SDN controller platforms. First, the lessons learned from the previous work on software ageing and rejuvenation were discussed, and a comprehensive overview of the possible alternatives in a design of such framework were presented. Next, the empirical data reported in public issue repositories was analyzed, localizing the common sources of software ageing, as well as their impact on the system performance, and repair strategies. Then, an experimental setup, including the testbed designed to measure the memory-leak profiles was presented. Finally, a design and optimization of software rejuvenation policies was discussed.

This is the first work addressing the issues of software ageing and rejuvenation in the network software, such as SDN orchestration platforms.

6.7.2 Discussion

Threats to validity. The main threats to the validity of the analysis presented in this chapter is the representativeness of the systems under study. Most of the empirical studies on software ageing are conducted for software written in memory-managed languages, such as `Java` (also `Python` and `C#`), in which garbage collection takes care of the memory disposal. In such programming languages, the memory-leaks occur because Garbage Collector is unable to remove unused objects, which are referenced somewhere else in the code. However, in languages such as `C/C++`, the programmer is responsible for memory de-allocation and explicit object disposal. A follow-up study will compare the results obtained for `Java`-based ONOS and ODL with *OpenContrail* (recently renamed as *Tungsten Fabric*), an open source production-grade SDN platform written mainly in `C++`, to study if an explicit memory management makes it more resilient to the effects of software ageing.

Generalization of the results. The analysis presented in this chapter exposed the prevalence of ageing related defects in the largest production grade controllers, ONOS and ODL, stemming not only from bugs but also inherent trade-off(s) in the design of distributed systems. Consequently, the results apply to all commercial solutions built on top of the code base of these two controllers. Moreover, results also apply to other distributed systems using the same distributed backend implementations, i.e., Atomix [85] and Akka [84].

Future work. The framework presented in this chapter is only a first step towards a better understanding of the effects of ageing on softwarized networks. The framework has proposed a memory-leak profiling for a network control system under controlled stress tests, while localization of the memory-leak sources has been performed by manual inspection of the previous outages. The framework can be further extended to account for the ageing profiles of network control systems "in the wild", as well as the automated localization of ageing defects.

Automated ageing trend detection in systems with variable workload. The network control systems in production environment may experience highly variable workload, both, in terms of intensity and service mix. Even when the resource-leak profiles are known for every operational point, reliable TTE

prediction is hard in such a setup. Different approaches to automated detection of resource leaks, based on trend analysis and state-based performability models should be designed and evaluated.

Localization of ageing-related bugs with differential code analysis and regression tests. An efficient localization of ageing defects in yet untested software releases is an active research area. Learning from mistakes, i.e., previous ageing related bugs, can improve the current testing practices, so that such bugs do not reoccur in new software releases. Localization of new ageing-related bugs can be done by means of regression testing, and differential code analysis [52, 116].

Chapter 7

Conclusions and Outlook

This chapter concludes the dissertation, with the summary and discussion of the key findings, as well as the outlook for future work and open research questions.

7.1 Summary and Discussion

Key findings. The main goal of this thesis was to advance the SoA understanding of the dependability assurance in softwarized networks, focusing on the requirements of mission critical applications, such as industrial networks. The following limitations of the related work on dependability assessment and assurance, defined in Fig. 1.2, have been addressed:

Attributes: The temporal variations of network software reliability have been studied for two different SDN orchestration platforms. Behavioural patterns, such as reliability growth due to the software maturity (Chapter 4) and reliability degradation due to software ageing (Chapter 6), are not precisely described in generic dependability metrics, such as availability and reliability. New attributes, such as *software maturity* and *network ageing*, have been proposed. Furthermore, *user-perceived service availability* has been analyzed (Chapter 5), to quantify the impact of network control plane failures on services and applications relying on such networks.

Threats: The extensive analysis of the threats, i.e., faults, errors and failures, for the two largest open source SDN orchestration platforms, ONOS and ODL, has been conducted, based on the software defects reported in their public bug repositories. The core SDN controller functions have been identified as the most critical ones, with the highest number of the recent defects belonging to the clustering module, which supports the implementation of the distributed control plane. A taxonomy and comprehensive analysis of the bugs in imperfect distributed SDN control plane implementations has been provided, as well as the corresponding modelling abstractions (Chapter 5). A particular class of defects related to scalability and performance issues in distributed network control plane, namely software ageing failures, have been experimentally validated (Chapter 6).

Means: The principal means of *fault tolerance* in legacy networks is the redundancy, which is often inefficient against the software failures due to the high degree of correlation between software failures, e.g., semantic software bugs. Moreover, the replication of stateful network functions requires tight coordination and synchronization, realized with distributed protocols, which inherently introduces new failure modes. The framework for assessment of dependability of real-life

distributed SDN implementations has been designed and evaluated in Chapter 5. The thesis also proposes high-fidelity stochastic models, able to reproduce the behaviour of real-life SDN orchestration platform, facilitating a reliable *fault forecasting*. A long term fault forecasting framework based on SRGM at the level of software release has been discussed Chapter 4. The forecasting of the failures at a shorter time scale and on the level of a single instance has been proposed in Chapter 6, together with a corresponding *fault prevention*, i.e., software rejuvenation, policies. The *fault removal* has not been explicitly addressed in the scope of this thesis. However, the comprehensive analysis of defects in the implementation of distributed control plane and software ageing defects, proposed in this work, can aid developers to gain deeper understanding of software defect patterns, improving the efficiency of existing testing frameworks, resulting in more efficient methods of fault removal.

Threats to validity. The main threats to validity of the proposed data-driven dependability assessment and assurance frameworks, discussed in detail in the corresponding chapters, can be summarized as follows:

- *Dataset accuracy and completeness*: the dependability analysis presented in this dissertation makes an extensive use of public software repositories of open source network components. However, the accuracy and completeness of open bug repositories cannot be guaranteed. Nevertheless, such repositories provide a good insight into production-grade network software and real-life incidents, facilitating a reasonable estimation of dependability bottlenecks.
- *Abstraction level*: the modelling abstractions used in this thesis, i.e., SRGM (Chapter 4), SRN (Chapter 5) and `workload-ageing` function (Chapter 6), come with inherent assumptions, representing a simplified view of the real-life software components. The chosen modelling formalism and level of abstraction represent a reasonable trade-off between the approximation loss and generalization power.
- *Representativeness of studied network components and use case scenarios*: Due to the time limitation, not all network components in softwarized network architectures could be studied. The focus was on SDN orchestration platforms, which have a central role in coordination of SDN and NFV control functions, and data plane management.

Generalization of the results. The analysis of the incentives for softwarization of industrial networks, although focused on the particular case study of wind park communication networks, can be extrapolated to other industrial applications. In particular, a framework to translate the technological benefits (i.e., achieving the industrial grade of service with open and standardized softwarized network solutions), to tangible savings can be applied to industrial networks with a different cost structure. Furthermore, the frameworks for dependability assessment proposed in this dissertation are not limited to network control software, and could find broader application for the assessment of other complex systems, with dynamic release cycles or distributed implementations.

The expected impact. The key results presented in this thesis have been disseminated to the wider audience, including standardization bodies (Internet Engineering Task Force (IETF) [18]), open source networking community (ONF [20]), and academia (book chapter in [1]), with the goal to extend the impact of key findings to network practitioners and relevant stakeholders.

IETF report [18]: E. Grossman et al. *Deterministic Networking Use Cases.* Internet-Draft draft-ietf-detnetuse-cases-20. Work in Progress. Internet Engineering Task Force, Dec. 2018.

ONF report [20]: S. Secci. *Security and Performance Comparison of ONOS and ODL controllers.* Informational report. 2019.

Book chapter [1]: B. Helvik, P. Vizarreta, C. Mas Machuca, P. Heegaard, and K. Trivedi. "*Dependability of Network Control Software.*" In: Resilient communication services protecting end-user applications from disaster-based failures (RECODIS). Ed. by J. Rack and D. Hutchison . Springer, 2019.

The proposed frameworks for dependability assessment of softwarized networks, presented in this thesis are only the first step towards a robust certification of softwarized networks. The open research questions and outlook for future work are discussed in the following section.

7.2 Outlook for the Future Work

The main research directions for the future (illustrated in Fig. 7.1), which have been discussed in detail in the previous chapters, are summarized here:

i. A techno-economic study on the incentives for softwarization of industrial networks presented in this thesis focused on local industrial networks. This work can be extended to support more complex scenarios, such as massive machine type communication in WSN, and QoS-aware orchestration of geo-distributed industrial network slices, such as Smart Grids.

ii. The SRGM-based framework for forecasting of the software defects in network control software can be further enhanced by improving the predictive power of reliability growth models using ANN. Moreover, an automated classification and analysis of the large corpus of software bugs with a network domain specific vocabulary, using NLP techniques, is another foreseen enhancement of the framework proposed in this work.

iii. The dependability assurance framework for distributed SDN platforms can be extended to support cross-layer optimization, in scenarios where physical network failures cannot be neglected. Moreover, the enhancements of test-suites that would effectively prevent the reoccurring failure patterns in implementation of distributed control plane, in particular ageing-related defects, are required.

iv. The framework for detection, characterization and prevention of the software ageing bugs can be further extended to account for the ageing profiles of network control systems "in the wild", as well as the automated localization of ageing defects.

The general trend in the presented future research directions can be observed: the ever-increasing role of machine learning in Software Reliability Engineering (SRE). The machine learning techniques, such as ANN and NLP, have been identified as helpful tools to improve the efficiency and accuracy of the classical SRE methods for failure forecasting, anomaly detection, automated defect classification, problem inference and knowledge extraction.

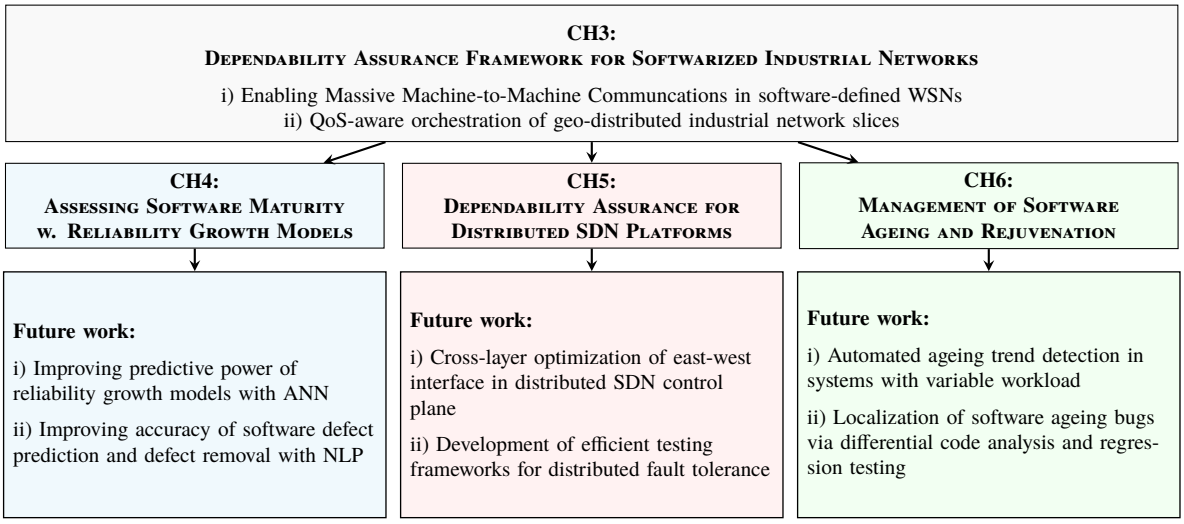


Figure 7.1: Outlook for the future work and open research questions in the area of dependability assurance for softwarized industrial networks.

Appendices

Appendix A

Mapping of Software Defects

A.1 Defects in Distributed SDN Implementations

This section contains the defects in distributed SDN implementations (Chapter 5), grouped into four classes: defects in implementation of distributed protocols (Tab. A.1), scalability and performance (Tab. A.2), high availability (Tab. A.3) and operational issues (Tab. A.4).

Table A.1: Defects in the implementation of distributed protocols (DP).

Alias	Bug key	Short description from bug repository
dp1	ONOS-7705	Only master of a device sees correct flow count
dp2	ONOS-7726	Links disappear after balancing masters
dp3	ONOS-7623	Device events are not replicated to other instances
dp4	ONOS-2121	Some ConsistentMap update operations do not publish events
dp5	ONOS-1883	Links disappear when devices change master
dp6	ONOS-436	Hosts learned via Gossip sometimes are missing ips
dp7	ONOS-4423	In releaxedConsistencyMode it is possible for a ConsistentMap instance to be out of sync indefinitely
dp8	CONTROLLER-1630	Follower not sync'ing up after rejoining cluster
dp9	CONTROLLER-1755	RaftActor lastApplied index moves backwards
dp10	CONTROLLER-1580	sal-remoterpc-connector: do not use calendar time for Bucket versions
dp11	CONTROLLER-1735	Entityownership leastload policy doesn't work normally
dp12	CONTROLLER-1717	RequestTimeoutException due to "Failed to transfer leadership" after become-prefix-leader with RoleChangeNotification not delivered
dp13	ONOS-4515	Cluster Device Role States out of Sync
dp14	ONOS-4529	intermittently OVS1.3 device lost mastership
dp15	ONOS-1400	BGPRouter crashes while kryo serialization due to recent change of distributed group store
dp16	CONTROLLER-1572	ReadDataReply Message was too large can result in "Received UnreachableMember" in cluster

Table A.2: Defects related to scalability and performance issues (SP).

Alias	Bug key	Short description from bug repository
sp1	CONTROLLER-1703	Tweak Akka and Java timeouts to a reasonable compromise between stability and failure detection
sp2	ONOS-356	Timeout in OpenFlowProvider when Intstalling large number of Intents
sp3	ONOS-2106	with a 625-sw topo, balance master does not balance well
sp4	ONOS-581	Chordal ring topology does not converge on ONOS until ONOS restart
sp5	ONOS-4785	Potential data loss during cluster scaling
sp6	ONOS-4567	Max number of intents install is less with cluster than standalone
sp7	ONOS-5279	Resource reservation takes too long in multi node cluster
sp8	ONOS-7024	Atomix 2.x timeouts
sp9	ONOS-6859	ResourceStore opens new Raft session on each transaction
sp10	ONOS-7382	Memory leak in ECFlowRuleStore
sp11	ONOS-6205	Memory leaks in DistributedMeterStore
sp12	ONOS-7412	Memory leaks in NettyMessagingManager
sp13	ONOS-3531	GossipApplicationStore throws StackOverflowError
sp14	OPNFLWPLUG-962	Multiple "expired" flows take up the memory resource of CONFIG DS which leads to Controller shutdown.
sp15	ONOS-4212	Memory leak problem when running CHO test

Table A.3: Defects related to high availability (HA).

Alias	Bug key	Short description from bug repository
ha1	ONOS-6149	Not able to configure heartbeatInterval and phiFailureThreshold properties in DistributedClusterStore
ha2	ONOS-7754	Configuration change causes false positives in failure detectors
ha3	ONOS-7755	False positives in failure detection when applying initial cluster configuration
ha4	ONOS-6682	Cluster becomes unavailable after a node becomes unavailable
ha5	ONOS-5992	ONOS HA cluster failure
ha6	ONOS-5347	ONOS cluster not able to recover after killing one of cluster member
ha7	ONOS-7528	Limit memory/CPU usage when Raft partitions are overloaded
ha8	ONOS-3423	When ONOS gets an out of memory exception it essentially becomes a zombie
ha9	ONOS-1673	Fail fast when DatabaseManager does not start up cleanly
ha10	ONOS-7586	ONOS leadership change does not occurs sometimes.
ha11	CONTROLLER-1693	UnreachableMember during remove-shard-replica prevents new leader to get elected
ha12	ONOS-1883	Links disappear when devices change master
ha13	CONTROLLER-1491	Entity Ownership Service: support graceful state handoff
ha14	ONOS-6042	Flows are not getting persisted after enabling the "persistenceEnabled" flag
ha15	ONOS-5690	Intent Persistence can't be enabled in ONOS
ha16	CONTROLLER-1794	Controller fails to join cluster
ha17	CONTROLLER-1630	Follower not sync'ing up after rejoining cluster
ha18	ONOS-1965	Deadlock can occur when a old candidate restarts and does not re-enter ledership race
ha19	ONOS-2015	Some devices have no ports after ONOS cluster restart

Table A.4: Defects related to operational issues (OP).

Alias	Bug key	Short description from bug repository
op1	CONTROLLER-1385	Make manual-down the default for akka-cluster
op2	CONTROLLER-1581	Clustering: Maintain a script to generate default akka configuration for multinode CSIT tests.
op3	CONTROLLER-1420	Clustering: Add a count field to stress-test RPC in car yang model
op4	CONTROLLER-779	Add test-case to check Install Snapshot functionality is handled correctly
op5	ONOS-7213	New cluster configuration cannot be serialized to JSON on configuration change
op6	ONOS-3453	Bundles not loaded in all nodes in a cluster
op7	ONOS-6647	Cluster formation using docker + kubernetes
op8	ONOS-7219	Single node ONOS from Docker image can't read cluster metadata
op9	ONOS-6401	ONOS nodes timeout when trying to connect to the cluster in vm test cluster
op10	ONOS-7436	Port latency and switch latency up/down went up dramatically after atomix 2.0.14

A.2 Defects Related to Software Ageing in SDN Controllers

This section presents the mapping of *aliases for software defects* (used in Chapter 6) to their respective keys used in Jira repositories are presented in (Tab. A.5).

Table A.5: Defects related to software ageing in SDN controllers.

Alias	Bug key	Short description from bug repository
PerfReg-1	ONOS-1339	IntentPerfInstaller overall rate decreases during 5min TP test, when in cluster mode
Storage-1	ONOS-7024	Atomix 2.x timeouts
Storage-2	ONOS-5179	StorageException when bringing up/down devices in CHO test
FileHandle-1	ONOS-7778	"Too many open files" when handling large Mininet topology
Session-1	ONOS-7889	Memory Leak in election leader
Session-2	ONOS-6859	ResourceStore opens new Raft session on each transaction
TxLeak-1	NETVIRT-883	Umbrella parent issue for grouping all suspected transaction leaks
TxLeak-2	CONTROLLER-1756	OOM due to huge Map in ShardDataTree
TxLeak-3	CONTROLLER-1746	OOM with large number of closed transactions
TxLeak-4	CONTROLLER-1762	ODL is up and ports are listening but not functional
DeadLeak-1	ONOS-7918	Netconf protocol handler doesn't release reference to replies
DeadLeak-2	ONOS-5172	ONOS intermittently failed to remove intent in CHO test
DeadLeak-3	CONTROLLER-1757	Singleton leader chasing exhausts heap space in few hours
DeadLeak-4	BGPCEP-631	Memory holdup in CachingImportPolicy
ExceptLeak-1	ONOS-6205	Memory leaks in DistributedMeterStore
ExceptLeak-2	ONOS-1441	DistributedFlowRuleStore doesn't remove flows for disconnected devices
ExceptLeak-3	ONOS-3531	GossipApplicationStore throws StackOverflowError
ExceptLeak-4	ONOS-6266	'garbageCollect' is not working for groups after ONOS restart
DesignIssue-1	ONOS-4212	Memory leak problem when running CHO test
DesignIssue-2	OPNFLWPLUG-962	Multiple "expired" flows take up the memory resource of CONFIG DS which leads to Controller shutdown.

Bibliography

Publications by the author

Book chapter

[1] B. Helvik, P. Vizarreta, C. Mas Machuca, P. Heegaard, and K. Trivedi. "Dependability of Network Control Software." In: *Resilient Communication Services Protecting End-user Applications from Disaster-based Failuresl*. Ed. by J. Rack and D. Hutchison. Springer: under preparation, 2019.

[2] C. Mas Machuca, F. Musumeci, P. Vizarreta, D. Pezaros, S. Jouet, M. Tornatore, et al. "Designing Reliable SDN Solutions." In: *Resilient Communication Services Protecting End-user Applications from Disaster-based Failuresl*. Ed. by J. Rack and D. Hutchison. Springer: under preparation, 2019.

Journal publications

[3] A. Papa, T. de Cola, P. Vizarreta, M. He, C. Mas Machuca, and W. Kellerer. "Design and Evaluation of Reconfigurable SDN LEO Constellations." In: *IEEE Transactions on Network and Service Management: under review* (2019).

[4] P. Vizarreta, C. Sieber, A. Van Bemten, A. Blenk, V. Ramachandra, W. Kellerer, C. Mas Machuca, and K. Trivedi. "ARES: A Framework for Management of Software Ageing and Rejuvenation in SDN." In: *IEEE Transactions on Network and Service Management: under review* (2019).

[5] P. Vizarreta, K. Trivedi, B. Helvik, P. Heegaard, A. Blenk, W. Kellerer, and C. Mas Machuca. "Assessing the Maturity of SDN Controllers with Software Reliability Growth Models." In: *IEEE Transactions on Network and Service Management* 15.3 (2018), pp. 1090–1104.

[6] P. Vizarreta, K. Trivedi, V. Mendiratta, W. Kellerer, and C. Mas Machuca. "DASON: Dependability Assessment Framework for Imperfect Distributed SDN Implementations." In: *IEEE Transactions on Network and Service Management: under review* (2019).

[7] P. Vizarreta, A. Van Bemten, E. Sakic, N. Petropolis, K. Abassi, W. Kellerer, and C. Mas Machuca. "Incentives for a Softwarization of Wind Park Communication Networks." In: *IEEE Communications Magazine* (2019), pp. 138–144.

Conference publications

[8] K. Fysarakis, N. E. Petroulakis, A. Roos, K. Abbasi, P. Vizarreta, G. Petropoulos, E. Sakic, G. Spanoudakis, and I. Askoxylakis. "A Reactive Security Framework for Operational Wind Parks Using Service Function Chaining." In: *IEEE Symposium on Computers and Communications*. IEEE. 2017, pp. 663–668.

[9] C. Mas Machuca, S. Secci, P. Vizarreta, F. Kuipers, A. Gouglidis, D. Hutchison, S. Jouet, D. Pezaros, A. Elmokashfi, P. Heegaard, et al. "Technology-related disasters: A survey towards disaster-resilient software defined networks." In: *IEEE International Workshop on Resilient Networks Design and Modeling*. IEEE. 2016, pp. 35–42.

[10] A. Papa, T. de Cola, P. Vizarreta, M. He, C. Mas Machuca, and W. Kellerer. "Dynamic SDN Controller Placement in a LEO Constellation Satellite Network." In: *IEEE Global Communications Conference*. IEEE. 2018.

[11] A. Van Bemten, J. W. Guck, P. Vizarreta, C. Mas Machuca, and W. Kellerer. "LARAC-SN and Mole in the Hole: Enabling Routing through Service Function Chains." In: (2018), pp. 1–5.

[12] P. Vizarreta. "Modelling, Design and Optimization of Dependable Softwarized Networks for Industrial Applications." In: *IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2018, pp. 170–173.

[13] P. Vizarreta, M. Condoluci, M. C. Machuca, T. Mahmoodi, and W. Kellerer. "QoS-driven Function Placement Reducing Expenditures in NFV Deployments." In: *IEEE International Conference on Communications*. IEEE. 2017, pp. 1–7.

[14] P. Vizarreta, P. Heegaard, B. Helvik, W. Kellerer, and C. Mas Machuca. "Characterization of Failure Dynamics in SDN Controllers." In: *IEEE International Workshop on Resilient Networks Design and Modeling*. IEEE. 2017, pp. 1–7.

[15] P. Vizarreta, C. Mas Machuca, and W. Kellerer. "Controller Placement Strategies for a Resilient SDN Control Plane." In: *IEEE International Workshop on Resilient Networks Design and Modeling*. IEEE. 2016, pp. 1–7.

[16] P. Vizarreta, E. Sakic, W. Kellerer, and C. Mas Machuca. "Mining Software Repositories for Predictive Modelling of Software Defects in SDN Controllers." In: *IFIP/IEEE International Symposium on Integrated Network Management*. IEEE. 2019, pp. 80–88.

[17] P. Vizarreta, K. Trivedi, B. Helvik, P. Heegaard, W. Kellerer, and C. Mas Machuca. "An Empirical Study of Software Reliability in SDN Controllers." In: *IEEE International Conference on Network and Service Management*. IEEE. 2017, pp. 1–9.

Technical reports

The full list of contributors is omitted due to the space limitations; only the principal editor is mentioned.

[18] E. Grossman. *Deterministic Networking Use Cases*. Internet-Draft draft-ietf-detnet-use-cases-20. Work in Progress. Internet Engineering Task Force, Dec. 2018. 88 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-detnet-use-cases-20.

[19] C. M. Machuca and P. Vizarreta. *Techno-economic Framework and Cost Models*. VirtuWind project deliverable D2.4. 2016. URL: http://www.virtuwind.eu/.

[20] S. Secci. *Security and Performance Comparison of ONOS and ODL controllers*. Informational report –. 2019. URL: https://www.opennetworking.org/wp-content/uploads/2019/09/ONOSvsODL-report-4.pdf.

General publications

[21] M. A. Ahmed and Y.-C. Kim. "Network Modeling and Simulation of Wind Power Farm with Switched Gigabit Ethernet." In: *International Symposium on Communications and Information Technologies*. IEEE. 2012, pp. 1009–1014.

[22] F. Alencar, M. Santos, M. Santana, and S. Fernandes. "How Software Aging Affects SDN: A View on the Controllers." In: *Global Information Infrastructure and Networking Symposium (GIIS), 2014*. IEEE. 2014, pp. 1–6.

[23] L. Andrade, M. Borba, A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém. "On the Benchmarking Mainstream Open Software-defined Networking Controllers." In: *ACM Latin America Networking Conference*. ACM. 2016, pp. 9–12.

[24] J. Araujo, R. Matos, V. Alves, P. Maciel, F. Souza, K. S. Trivedi, et al. "Software Aging in the Eucalyptus Cloud Computing Infrastructure: Characterization and Rejuvenation." In: *ACM Journal on Emerging Technologies in Computing Systems* 10.1 (2014), p. 11.

[25] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker. "Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure." In: *ACM Middleware Industry Track Workshop*. ACM. 2011, p. 4.

[26] F. Bannour, S. Souihi, and A. Mellouk. "Distributed SDN Control: Survey, Taxonomy, and Challenges." In: *Communications Surveys & Tutorials* 20.1 (2017), pp. 333–354.

[27] K. Basu, M. Younas, A. W. W. Tow, and F. Ball. "Performance Comparison of a SDN Network between Cloud-Based and Locally Hosted SDN Controllers." In: *International Conference on Big Data Computing Service and Applications*. IEEE. 2018, pp. 49–55.

[28] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. "ONOS: Towards an Open, Distributed SDN OS." In: *ACM Workshop on Hot topics in Software Defined Networking*. ACM. 2014, pp. 1–6.

[29] V. Bhuvaneswaran, A. Basil, M. Tassinari, V. Manral, and S. Banks. *Benchmarking Methodology for Software-Defined Networking (SDN) Controller Performance*. Tech. rep. RFC-8456. Internet Engineering Task Force, Oct. 2018. URL: http://www.rfc-editor.org/pdfrfc/rfc8456.txt.pdf.

[30] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen. "Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics." In: *International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 1–12.

[31] A. Blenk, A. Basta, L. Henkel, J. Zerwas, W. Kellerer, and S. Schmid. "perfbench: A tool for Predictability Analysis in Multi-tenant Software-Defined Networks." In: *SIGCOMM Posters and Demos*. ACM. 2018, pp. 66–68.

[32] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo. "Memory Leak Analysis of Mission-critical Middleware." In: *Journal of Systems and Software* 83.9 (2010), pp. 1556–1567.

[33] CERIAS - CISCO Critical Infrastructure Assurance Group (CIAG). *SCADA honeynet project*. Oct. 18, 2017. URL: http://scadahoneynet.sourceforge.net/.

[34] S. Chandra and P. M. Chen. "Whither Generic Recovery From Application Faults? A Fault Study Using Open-source Software." In: *IEEE International Conference on Dependable Systems and Networks*. IEEE. 2000, pp. 97–106.

[35] C.-C. Chen, C.-T. Lin, H.-H. Huang, S.-W. Huang, and C.-Y. Huang. "CARATS: A Computer-aided Reliability Assessment Tool for Software Based on Object Oriented Design." In: *IEEE Region 10 Conference TENCON*. IEEE. 2006, pp. 1–4.

[36] *Communication Delivery Time Performance Requirements for Electric Power Substation Automation*. Tech. rep. IEEE Standard 1646-2004. 2004.

[37] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono. "Software Aging Analysis of the Android Mobile OS." In: *International Symposium on Software Reliability Engineering*. IEEE. 2016, pp. 478–489.

[38] D. Cotroneo, R. Natella, and R. Pietrantuono. "Predicting Aging-related Bugs Using Software Complexity Metrics." In: *Performance Evaluation* 70.3 (2013), pp. 163–178.

[39] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. "A Survey of Software Aging and Rejuvenation Studies." In: *ACM Journal on Emerging Technologies in Computing Systems* 10.1 (2014), p. 8.

[40] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. "Software Aging Analysis of the Linux Operating System." In: *International Symposium on Software Reliability Engineering*. IEEE. 2010, pp. 71–80.

[41] D. Cotroneo, S. Orlando, R. Pietrantuono, and S. Russo. "A Measurement-based Ageing Analysis of the JVM." In: *Software Testing, Verification and Reliability* 23.3 (2013), pp. 199–239.

[42] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders. "Möbius: An Extensible Tool for Performance and Dependability Modeling." In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2000, pp. 332–336.

[43] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano. "nDPI: Open-source High-speed Deep Packet Inspection." In: *International Wireless Communications and Mobile Computing Conference*. IEEE. 2014, pp. 617–622.

[44] C. Di Martino, U. Giordano, N. Mohanasamy, S. Russo, and M. Thottan. “In Production Performance Testing of SDN Control Plane for Telecom Operators.” In: *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2018, pp. 642–653.

[45] ETSI GS NFV-REL 003 V1.1.1 (2016-04). *Network Functions Virtualisation (NFV); Reliability; Report on Models and Features for End-to-End Reliability*. 2016.

[46] J. Fan, Z. Ye, C. Guan, X. Gao, K. Ren, and C. Qiao. “GREP: Guaranteeing Reliability with Enhanced Protection in NFV.” In: *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM. 2015, pp. 13–18.

[47] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. *TCP Extensions for Multipath Operation with Multiple Addresses*. Internet-Draft draft-ietf-mptcp-rfc6824bis-12. Work in Progress. Internet Engineering Task Force, Oct. 2018. 80 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mptcp-rfc6824bis-12.

[48] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions.” In: *USENIX Conference on File and Storage Technologies*. 2017, pp. 149–166.

[49] W. Garcia and T. Benson. “A First Look at Bugs in OpenStack.” In: *ACM Workshop on Cloud-Assisted Networking*. ACM. 2016, pp. 67–72.

[50] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. S. Trivedi. “A Methodology for Detection and Estimation of Software Aging.” In: *International Symposium on Software Reliability Engineering*. IEEE. 1998, pp. 283–292.

[51] D. Georgakopoulos, P. P. Jayaraman, M. Fazia, M. Villari, and R. Ranjan. “Internet of Things and Edge Cloud Computing Roadmap for Manufacturing.” In: *IEEE Cloud Computing* 3.4 (2016), pp. 66–73.

[52] M. Ghanavati and A. Andrzejak. “Automated Memory Leak Diagnosis by Regression Testing.” In: *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2015, pp. 191–200.

[53] M. Ghanavati, A. Andrzejak, D. Costa, and J. Seboek. “Poster: Memory and resource leak defects Java projects: An Empirical Study.” In: *International Conference on Software Engineering*. IEEE. 2018.

[54] M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. “Memory and Resource Leak Defects in Java Projects: an Empirical Study.” In: *ACM International Conference on Software Engineering: Companion Proceeedings*. ACM. 2018, pp. 410–411.

[55] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi. “Scalable analytics for IaaS cloud availability.” In: *Transactions on Cloud Computing* 2.1 (2014), pp. 57–70.

[56] P. Gill, N. Jain, and N. Nagappan. “Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications.” In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 350–361.

[57] A. Goel and K. Okumoto. “Time-dependent Error Detection Rate Model for Software and Other Performance Measure.” In: *IEEE Transactions on Software Engineering* 11.12 (1985), p. 285.

[58] S. S. Gokhale, M. R. Lyu, and K. S. Trivedi. “Analysis of Software Fault Removal Policies Using a Non-homogeneous Continuous Time Markov Chain.” In: *Software Quality Journal* 12.3 (2004), pp. 211–230.

[59] S. S. Gokhale and K. S. Trivedi. “Log-logistic Software Reliability Growth Model.” In: *IEEE International High-Assurance Systems Engineering Symposium*. IEEE. 1998, pp. 34–41.

[60] A. J. Gonzalez, G. Nencioni, B. E. Helvik, and A. Kamisinski. “A Fault-Tolerant and Consistent SDN Controller.” In: *IEEE Global Communications Conference*. IEEE. 2016, pp. 1–6.

[61] A. Gonzalez, P. Gronsund, K. Mahmood, B. Helvik, P. Heegaard, and G. Nencioni. “Service Availability in the NFV Virtualized Evolved Packet Core.” In: *IEEE Global Communications Conference*. IEEE. 2015, pp. 1–6.

[62] K. Goseva-Popstojanova, A. P. Mathur, and K. S. Trivedi. “Comparison of Architecture-based Software Reliability Models.” In: *International Symposium on Software Reliability Engineering*. IEEE. 2001, pp. 22–31.

[63] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. “Evolve or Die: High-availability Design Principles Drawn from Google’s Network Infrastructure.” In: *ACM SIGCOMM*. ACM. 2016, pp. 58–72.

[64] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. “Analysis of Software Aging in a Web Server.” In: *IEEE Transactions on reliability* 55.3 (2006), pp. 411–420.

[65] J. Guck, A. Van Bemten, and W. Kellerer. “DetServ: Network Models for Real-Time QoS Provisioning in SDN-based Industrial Environments.” In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 1003–1017.

[66] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, et al. “What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems.” In: *ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[67] M. Gürsu, M. Vilgelm, E. Fazli, and W. Kellerer. “A Medium-access Approach to Wireless Technologies for Reliable Communication in Aircraft.” In: *Wireless Sensor Systems for Extreme Environments: Space, Underwater, Underground and Industrial* (2017), pp. 431–451.

[68] R. Hanmer, L. Jagadeesan, V. Mendiratta, and H. Zhang. “Friend or Foe: Strong Consistency vs. Overload in High-Availability Distributed Systems and SDN.” In: *International Symposium on Software Reliability Engineering*. IEEE. 2018, pp. 1–6.

[69] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, and M. Vechev. “SDNRacer: Concurrency Analysis for Software-Defined Networks.” In: *ACM SIGPLAN Notices*. Vol. 51. 6. ACM. 2016, pp. 402–415.

[70] S. Hassas Yeganeh and Y. Ganjali. “Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications.” In: *ACM Workshop on Hot topics in SDN*. ACM. 2012, pp. 19–24.

[71] N. Hayashibara, D. Xavier, R. Yared, T. Katayama, et al. “The φ Accrual Failure Detector.” In: *null*. IEEE. 2004, pp. 66–78.

[72] B. Heller, R. Sherwood, and N. McKeown. “The Controller Placement Problem.” In: *ACM Wworkshop on Hot topics in SDN*. ACM. 2012, pp. 7–12.

[73] J. G. Herrera and J. F. Botero. “Resource Allocation in NFV: A Comprehensive Survey.” In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532.

[74] C. Hirel, R. Sahner, X. Zang, and K. Trivedi. “Reliability and performability modeling using SHARPE 2000.” In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2000, pp. 345–349.

[75] C. Hirel, B. Tuffin, and K. S. Trivedi. “Spnp: Stochastic petri nets. version 6.0.” In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2000, pp. 354–357.

[76] C.-Y. Huang, T.-Y. Hung, and C.-J. Hsu. “Software Reliability Prediction and Analysis Using Queueing Models with Multiple Change-Points.” In: *IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE. 2009, pp. 212–221.

[77] C.-Y. Huang and M. R. Lyu. “Estimation and Analysis of Some Generalized Multiple Change-point Software Reliability Models.” In: *IEEE Transactions on Reliability* 60.2 (2011), pp. 498–514.

[78] C.-Y. Huang and M. R. Lyu. “Optimal release time for software systems considering cost, testing-effort, and test efficiency.” In: *IEEE Transactions on Reliability* 54.4 (2005), pp. 583–591.

[79] C.-Y. Huang, M. R. Lyu, and S.-Y. Kuo. “A Unified Scheme of Some Nonhomogenous Poisson Process Models for Software Reliability Estimation.” In: *IEEE Transactions on Software Engineering* 29.3 (2003), pp. 261–269.

[80] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. “Software Rejuvenation: Analysis, Module and Applications.” In: *International Symposium on Fault-Tolerant Computing*. IEEE. 1995, pp. 381–390.

[81] ICS-CERT. *Cyber-Attack Against Ukrainian Critical Infrastructure*. 2015. URL: https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01.

[82] *International Standard 61400-25: Communications for Monitoring and Control of Wind Power Plants*. Tech. rep. IEC 61400-25. 2017.

[83] Intracom Telecom. *Multithreaded CBench (MT-CBench)*. URL: https://github.com/intracom-telecom-sdn/mtcbench.

[84] J. Bonér (Lightbend). *Akka*. Version 2.5.21. Mar. 10, 2019. URL: https://akka.io/.

[85] J. Halterman (ONF). *Atomix*. Version 3.0. Mar. 10, 2019. URL: https://atomix.io/.

[86] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. “B4: Experience with a globally-deployed software defined WAN.” In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–14.

[87] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries. "A Flexible OpenFlow-controller Benchmark." In: *European Workshop on Software Defined Networking*. IEEE. 2012, pp. 48–53.

[88] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia. "OFCProbe: A platform-independent Tool for OpenFlow Controller Analysis." In: *International Conference on Communications and Electronics*. IEEE. 2014, pp. 182–187.

[89] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open Source Scientific Tools for Python*. 2001–. URL: http://www.scipy.org/.

[90] P. Kapur and R. Garg. "Optimal Sofware Release Policies for Software Reliability Growth Models under Imperfect Debugging." In: *RAIRO-Operations Research* 24.3 (1990), pp. 295–305.

[91] P. Kapur, H. Pham, S. Anand, and K. Yadav. "A Unified Approach for Developing Software Reliability Growth Models in the Presence of Imperfect Debugging and Error Generation." In: *IEEE Transactions on Reliability* 60.1 (2011), pp. 331–340.

[92] Z. K. Khattak, M. Awais, and A. Iqbal. "Performance Evaluation of OpenDaylight SDN Controller." In: *International Conference on Parallel and Distributed Systems*. IEEE. 2014, pp. 671–676.

[93] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou. "Feature-based Comparison and Selection of Software Defined Networking (SDN) Controllers." In: *World Congress on Computer Applications and Information Systems*. IEEE. 2014, pp. 1–7.

[94] D. S. Kim, F. Machida, and K. S. Trivedi. "Availability Modeling and Analysis of a Virtualized System." In: *IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2009, pp. 365–371.

[95] M. Kimura, T. Toyota, and S. Yamada. "Economic Analysis of Software Release Problems with Warranty Cost and Reliability Requirement." In: *Reliability Engineering & System Safety* 66.1 (1999), pp. 49–55.

[96] H. S. Koch and P. Kubat. "Optimal Release Time of Computer Software." In: *IEEE Transactions on Software Engineering* 3 (1983), pp. 323–327.

[97] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. "Onix: A Distributed Control Platform for Large-scale Production Networks." In: *USENIX Symposium on Operating Systems Design and Implementation*. Vol. 10. 2010, pp. 1–6.

[98] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-defined Networking: A Comprehensive Survey." In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[99] R. Lai, M. Garg, P. K. Kapur, and S. Liu. "A Study of When to Release a Software Product from the Perspective of Software Reliability Models." In: *Journal of Software* 6.4 (2011), pp. 651–661.

[100] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann. “Heuristic Approaches to the Controller Placement Problem in Large Scale SDN Networks.” In: *IEEE Transactions on Network and Service Management* 12.1 (2015), pp. 4–17.

[101] F. Langner and A. Andrzejak. “Detecting Software Aging in a Cloud Computing Framework by Comparing Development Versions.” In: *IFIP/IEEE International Symposium on Integrated Network Management*. IEEE. 2013, pp. 896–899.

[102] F. Langner and A. Andrzejak. “Detection and Root Cause Analysis of Memory-related Software Aging Defects by Automated Tests.” In: *International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE. 2013, pp. 365–369.

[103] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. “Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software.” In: *ACM workshop on architectural and system support for improving software dependability*. ACM. 2006, pp. 25–33.

[104] Linux Foundation. *OpenDaylight*. 2017. URL: https://www.opendaylight.org/.

[105] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. “Crystalnet: Faithfully Emulating Large Production Networks.” In: *ACM Symposium on Operating Systems Principles*. ACM. 2017, pp. 599–613.

[106] F. Longo, S. Distefano, D. Bruneo, and M. Scarpa. “Dependability Modeling of Software Defined Networking.” In: *Computer Networks* 83 (2015), pp. 280–296.

[107] S. Lu, S. Park, E. Seo, and Y. Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics.” In: *ACM SIGOPS Operating Systems Review* 42.2 (2008), pp. 329–339.

[108] M. R. Lyu et al. *Handbook of Software Reliability Engineering*. IEEE computer society press CA, 1996.

[109] M. R. Lyu and A. Nikora. “CASRE: A Computer-aided Software Reliability Estimation Tool.” In: *IEEE International Workshop on Computer-Aided Software Engineering*. IEEE. 1992, pp. 264–275.

[110] T. Mahmoodi, V. Kulkarni, W. Kellerer, P. Mangan, F. Zeiger, S. Spirou, I. Askoxylakis, X. Vilajosana, H. J. Einsiedler, and J. Quittek. “VirtuWind: Virtual and Programmable Industrial Network Prototype Deployed in Operational Wind Park.” In: *Transactions on Emerging Telecommunications Technologies* 27.9 (2016), pp. 1281–1288.

[111] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Diot. “Characterization of Failures in an IP Backbone.” In: *INFOCOM*. Vol. 4. IEEE. 2004, pp. 2307–2317.

[112] C. Mas Machuca, P. Vizarreta, R. Durner, D. Santos, and A. de Sousa. “Design Problems Towards Reliable SDN Networks.” In: *Photonic Networks and Devices*. Optical Society of America. 2018, NeM2F–1.

[113] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi. “A Systematic Differential Analysis for Fast and Robust Detection of Software Aging.” In: *IEEE International Symposium on Reliable Distributed Systems*. IEEE. 2014, pp. 311–320.

[114] R. Matias, P. A. Barbetta, K. S. Trivedi, and P. J. Freitas Filho. "Accelerated Degradation Tests Applied to Software Aging Experiments." In: *IEEE Transactions on reliability* 59.1 (2010), pp. 102–114.

[115] R. Matias and J. Paulo Filho. "An Experimental Study on Software Aging and Rejuvenation in Web Servers." In: *International Computer Software and Applications Conference*. Vol. 1. IEEE. 2006, pp. 189–196.

[116] R. Matias, G. O. de Sena, A. Andrzejak, and K. S. Trivedi. "Software Aging Detection Based on Differential Analysis: an Experimental Study." In: *IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2016, pp. 71–77.

[117] R. May, A. El-Hassany, L. Vanbever, and M. Vechev. "BigBug: Practical Concurrency Analysis for SDN." In: *ACM Symposium on SDN Research*. ACM. 2017, pp. 88–94.

[118] A. Medem, M.-I. Akodjenou, and R. Teixeira. "Troubleminer: Mining Network Trouble Tickets." In: *IFIP/IEEE International Symposium on Integrated Network Management - Workshops*. IEEE. 2009, pp. 113–119.

[119] J. Medved, R. Varga, A. Tkacik, and K. Gray. "Opendaylight: Towards a Model-driven SDN Controller Architecture." In: *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. IEEE. 2014, pp. 1–6.

[120] V. B. Mendiratta. "Reliability Analysis of Clustered Computing Systems." In: *IEEE International Symposium on Software Reliability Engineering*. IEEE. 1998, pp. 268–272.

[121] V. B. Mendiratta, L. J. Jagadeesan, R. Hanmer, and M. R. Rahman. "How Reliable Is My Software-Defined Network? Models and Failure Impacts." In: *International Symposium on Software Reliability Engineering Workshops*. IEEE. 2018, pp. 83–88.

[122] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. "Network Function Virtualization: State-of-the-art and Research Challenges." In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 236–262.

[123] J. Miserez, P. Bielik, A. El-Hassany, L. Vanbever, and M. Vechev. "SDNRacer: Detecting Concurrency Violations in Software-defined Networks." In: *ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 22.

[124] N. Mitchell and G. Sevitsky. "LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications." In: *European Conference on Object-Oriented Programming*. Springer. 2003, pp. 351–377.

[125] A. S. Muqaddas, P. Giaccone, A. Bianco, and G. Maier. "Inter-controller Traffic to Support Consistency in ONOS Clusters." In: *Transactions on Network and Service Management* 14.4 (2017), pp. 1018–1031.

[126] J. D. Musa and K. Okumoto. "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement." In: *International conference on Software Engineering*. IEEE Press. 1984, pp. 230–238.

[127] G. Nencioni, B. E. Helvik, A. J. Gonzalez, P. E. Heegaard, and A. Kamisinski. "Availability Modelling of Software-Defined Backbone Networks." In: *IEEE/IFIP International Conference on Dependable Systems and Networks - Workshop*. IEEE. 2016, pp. 105–112.

[128] G. Nencioni, B. E. Helvik, and P. E. Heegaard. "Including Failure Correlation in Availability Modeling of a Software-Defined Backbone Network." In: *Transactions on Network and Service Management* 14.4 (2017), pp. 1032–1045.

[129] Netflix. *SimianArmy: Chaos Monkey*. Apr. 24, 2019. URL: https://github.com/Netflix/chaosmonkey.

[130] A. Nguyen-Ngoc, S. Raffeck, S. Lange, S. Geissler, T. Zinner, and P. Tran-Gia. "Benchmarking the ONOS Controller with OFCProbe." In: *International Conference on Communications and Electronics*. IEEE. 2018, pp. 367–372.

[131] T. A. Nguyen, T. Eom, S. M. An, J. S. Park, J. B. Hong, and D. S. Kim. "Availability Modeling and Analysis for Software Defined Networks." In: *IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2015, pp. 159–168.

[132] OFLOPS. *CBench: A Benchmarking Tool for Controllers*. URL: https://github.com/mininet/oflops/tree/master/cbench.

[133] M. Ohba. "Software Reliability Analysis Models." In: *IBM Journal of research and Development* 28.4 (1984), pp. 428–443.

[134] K. Ohishi, H. Okamura, and T. Dohi. "Gompertz Software Reliability Model: Estimation Algorithm and Empirical Validation." In: *Journal of Systems and Software* 82.3 (2009), pp. 535–543.

[135] H. Okamura and T. Dohi. "A Generalized Bivariate Modeling Framework of Fault Detection and Correction Processes." In: *International Symposium on Software Reliability Engineering*. IEEE. 2017, pp. 35–45.

[136] K. Okumoto and A. L. Goel. "Optimum Release Time for Software Systems Based on Reliability and Cost Criteria." In: *Journal of Systems and Software* 1.4 (1980), pp. 315–318.

[137] ON.Lab. *ONOS: Open Neetwork Operating System*. 2017. URL: http://onosproject.org/.

[138] ONF. *OpenFlow Controller Benchmarking Methodologies*. Tech. rep. ONF TR-539. Open Networking Foundation, Nov. 2016. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/TR-539_OpenFlow_Controller_Benchmarking_Methodologies_v1.pdf.

[139] D. Ongaro and J. K. Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[140] ONOS. "SDN Control Plane Performance." In: *ONOS Project Whitepaper*. July 2017, pp. 1–23.

[141] Open source security. *pfSense*. Oct. 18, 2017. URL: https://www.pfsense.org/.

[142] S. Orlando. "Software Aging Analysis of Off-the-Shelf Software Items." PhD thesis. Università degli Studi di Napoli Federico II, 2008.

[143] S. Osaki. *Stochastic Models in Reliability and Maintenance*. Springer, 2002.

[144] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. "CAP for networks." In: *ACM SIGCOMM workshop on Hot topics in Software Defined Networking*. ACM. 2013, pp. 91–96.

[145] A. Pettener. "SCADA and communication networks for large scale offshore wind power systems." In: *Renewable Power Generation (RPG 2011), IET Conference on*. IET. 2011, pp. 1–6.

[146] H. Pham, L. Nordmann, and Z. Zhang. "A General Imperfect Software Debugging Model with S-shaped Fault Detection Rate." In: *IEEE Transactions on Reliability* 48.2 (1999), pp. 169–175.

[147] H. Pham and X. Zhang. "NHPP Software Reliability and Cost Models with Testing Coverage." In: *European Journal of Operational Research* 145.2 (2003), pp. 443–454.

[148] K. Phemius, M. Bouet, and J. Leguay. "Disco: Distributed Multi-domain SDN Controllers." In: *IEEE Network Operations and Management Symposium*. IEEE. 2014, pp. 1–4.

[149] R. Potharaju and N. Jain. "When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services." In: *Symposium on Cloud Computing*. ACM. 2013, p. 15.

[150] R. Potharaju, N. Jain, and C. Nita-Rotaru. "Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets." In: *USENIX Symposium on Networked Systems Design and Implementation*. 2013, pp. 127–141.

[151] N. Provos. "Honeyd - A Virtual Honeypot Daemon." In: *10th DFN-CERT Workshop, Hamburg, Germany*. Vol. 2. 2003, p. 4.

[152] F. Qin, Z. Zheng, Y. Qiao, and K. S. Trivedi. "Studying Aging-Related Bug Prediction Using Cross-Project Models." In: *IEEE Transactions on Reliability* 99 (2018), pp. 1–20.

[153] L. Qu, C. Assi, K. Shaban, and M. J. Khabbaz. "A Reliability-aware Network Service Chain Provisioning with Delay Guarantees in NFV-enabled Enterprise Datacenter Networks." In: *IEEE Transactions on Network and Service Management* 14.3 (2017), pp. 554–568.

[154] C. Rahmani, H. Siy, and A. Azadmanesh. "An Experimental Analysis of Open Source Software Reliability." In: *Department of Defense/Air Force Office of Scientific Research* (2009).

[155] D. S. S. Research. *Jepsen*. Version 9. Apr. 24, 2019. URL: https://github.com/jepsen-io/.

[156] M. Roesch et al. "Snort: Lightweight Intrusion Detection for Networks." In: *Lisa*. Vol. 99. 1. 1999, pp. 229–238.

[157] F. J. Ros and P. M. Ruiz. "Five Nines of Southbound Reliability in Software-Defined Networks." In: *ACM Workshop on Hot topics in Software Defined Networking*. ACM. 2014, pp. 31–36.

[158] B. Rossi, B. Russo, and G. Succi. "Modelling Failures Occurrences of Open Source Software with Reliability Growth." In: *IFIP International Conference on Open Source Systems*. Springer. 2010, pp. 268–280.

[159] E. Sakic, N. Deric, and W. Kellerer. "MORPH: An Adaptive Framework for Efficient and Byzantine Fault-Tolerant SDN Control Plane." In: *Journal on Selected Areas in Communications* (2018), pp. 1–13.

[160] E. Sakic and W. Kellerer. "Impact of Adaptive Consistency on Distributed SDN Applications: An Empirical Study." In: *Journal on Selected Areas in Communications* (), pp. 1–13.

[161] E. Sakic and W. Kellerer. “Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane.” In: *IEEE Transactions on Network and Service Management* 15.1 (2017), pp. 304–318.

[162] E. Sakic, V. Kulkarni, V. Theodorou, A. Matsiuk, S. Kuenzer, N. E. Petroulakis, and K. Fysarakis. “VirtuWind–An SDN-and NFV-based Architecture for Softwarized Industrial Networks.” In: *International Conference on Measurement, Modelling and Evaluation of Computing Systems*. Springer. 2018, pp. 251–261.

[163] D. Santos, A. de Sousa, and C. Mas Machuca. “Combined Control and Data Plane Robustness of SDN Networks against Malicious Node Attacks.” In: *IEEE International Conference on Network and Service Management*. IEEE. 2018, pp. 54–62.

[164] T. Sato, S. Ata, I. Oka, and Y. Sato. “Abstract Model of SDN Architectures Enabling Comprehensive Performance Comparisons.” In: *International Conference on Network and Service Management*. IEEE. 2015, pp. 99–107.

[165] T. Sauter. “The Three Generations of Field-level Networks—Evolution and Compatibility Issues.” In: *IEEE Transactions on Industrial Electronics* 57.11 (2010), pp. 3585–3595.

[166] B. Schroeder and G. A. Gibson. “Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?” In: *USENIX Conference on File and Storage Technologies*. Vol. 7. 1. 2007, pp. 1–16.

[167] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi. “An Architectural Evaluation of SDN Controllers.” In: *International Conference on Communications*. IEEE. 2013, pp. 3504–3508.

[168] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. “Advanced Study of SDN/OpenFlow Controllers.” In: *Central & Eastern European Software Engineering Conference in Russia*. ACM. 2013, p. 1.

[169] L. Silva, H. Madeira, and J. G. Silva. “Software Aging and Rejuvenation in a SOAP-based Server.” In: *IEEE International Symposium on Network Computing and Applications*. IEEE. 2006, pp. 56–65.

[170] D. Suh, S. Jang, S. Han, S. Pack, T. Kim, and J. Kwak. “On Performance of OpenDaylight Clustering.” In: *NetSoft Conference and Workshops*. IEEE. 2016, pp. 407–410.

[171] H. Sukhwani, J. M. Martinez, X. Chang, K. S. Trivedi, and A. Rindos. “Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric).” In: *Symposium on Reliable Distributed Systems*. IEEE. 2017, pp. 253–255.

[172] H. Sukhwani, R. Matias Jr, K. S. Trivedi, and A. Rindos. “Monitoring and Mitigating Software Aging on IBM Cloud Controller System.” In: *IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2017, pp. 266–272.

[173] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos. “Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network).” In: *International Symposium on Network Computing and Applications*. IEEE. 2018, pp. 1–8.

[174] *Supervisory Control And Data Acquisition (SCADA) and Automation Systems*. Tech. rep. IEEE Standard C37.1-2007. 2007.

[175] S. M. Syed-Mohamad and T. McBride. "Reliability Growth of Open Source Software Using Defect Analysis." In: *International Conference on Computer Science and Software Engineering*. Vol. 2. IEEE. 2008, pp. 662–667.

[176] A. Tootoonchian and Y. Ganjali. "Hyperflow: A Distributed Control Plane for OpenFlow." In: *Internet network management conference on Research on enterprise networking*. 2010, pp. 3–3.

[177] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. "On Controller Performance in Software-Defined Networks." In: *Hot-ICE* 12 (2012), pp. 1–6.

[178] K. S. Trivedi. "SREPT: A Tool for Software Reliability Estimation and Prediction." In: *International Conference on Dependable Systems and Networks*. IEEE. 2002, p. 546.

[179] K. S. Trivedi and A. Bobbio. *Reliability and Availability Engineering: Modeling, Analysis, and Applications*. Cambridge University Press, 2017.

[180] N. Ullah, M. Morisio, and A. Vetro. "A Comparative Analysis of Software Reliability Growth Models Using defects Data of Closed and Open Source Software." In: *IEEE Software Engineering Workshop*. IEEE. 2012, pp. 187–192.

[181] K. Vaidyanathan and K. S. Trivedi. "A Comprehensive Model for Software Rejuvenation." In: *IEEE Transactions on Dependable and Secure Computing* 2.2 (2005), pp. 124–137.

[182] N. A. Valentim, A. Macedo, and R. Matias. "A Systematic Mapping Review of the First 20 Years of Software Aging and Rejuvenation Research." In: *IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE. 2016, pp. 57–63.

[183] J. Vestin, A. Kassler, and J. Akerberg. "Resilient Software Defined Networking for Industrial Control Networks." In: *International Conference on Information, Communications and Signal Processing*. IEEE. 2015, pp. 1–5.

[184] M. Vilgelm, S. Schiessl, H. Al-Zubaidy, W. Kellerer, and J. Gross. "On the Reliability of LTE Random Access: Performance Bounds for Machine-to-Machine Burst Resolution Time." In: *IEEE International Conference on Communications*. IEEE. 2018, pp. 1–7.

[185] S. A. Vilkomir, D. L. Parnas, V. B. Mendiratta, and E. Murphy. "Availability Evaluation of Hardware/Software Systems with Several Recovery Procedures." In: *IEEE International Computer Software and Applications Conference*. Vol. 1. IEEE. 2005, pp. 473–478.

[186] D. Wang and K. S. Trivedi. "Modeling User-perceived Service Availability." In: *International Service Availability Symposium*. Springer. 2005, pp. 107–122.

[187] K. Wang, H. Li, S. Maharjan, Y. Zhang, and S. Guo. "Green Energy Scheduling for Demand Side Management in the Smart Grid." In: *IEEE Transactions on Green Communications and Networking* 2.2 (2018), pp. 596–611.

[188] M. Wei and Z. Chen. "Study of LANs Access Technologies in Wind Power System." In: *IEEE Power and Energy Society General Meeting*. IEEE. 2010, pp. 1–6.

[189] Y. Wu, Q. Hu, M. Xie, and S. H. Ng. "Modeling and Analysis of Software Fault Detection and Correction Process by Considering Time Dependency." In: ().

[190] W. Xie, Y. Hong, and K. S. Trivedi. “Software Rejuvenation Policies for Cluster Systems Under Varying Workload.” In: *IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2004, pp. 122–129.

[191] G. Xu, M. D. Bond, F. Qin, and A. Rountev. “LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks.” In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 270–282.

[192] S. Yamada, M. Ohba, and S. Osaki. “S-shaped Reliability Growth Modeling for Software Error Detection.” In: *IEEE Transactions on Reliability* 32.5 (1983), pp. 475–484.

[193] S. Yamada, H. Ohtera, and H. Narihisa. “Software Reliability Growth Models with Testing-effort.” In: *IEEE Transactions on Reliability* 35.1 (1986), pp. 19–23.

[194] S. Yamada and S. Osaki. “Cost-reliability Optimal Release Policies for Software Systems.” In: *IEEE Transactions on Reliability* 34.5 (1985), pp. 422–424.

[195] S. Yamada, T. Ichimori, and M. Nishiwaki. “Optimal Allocation Policies for Testing-resource Based on a Software Reliability Growth Model.” In: *Mathematical and Computer Modelling* 22.10-12 (1995), pp. 295–301.

[196] D. Yan, G. Xu, S. Yang, and A. Rountev. “LeakChecker: Practical Static Memory Leak Detection for Managed Languages.” In: *IEEE/ACM International Symposium on Code Generation and Optimization*. ACM. 2014, p. 87.

[197] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, et al. “Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering.” In: *Conference of the ACM Special Interest Group on Data Communication*. ACM. 2017, pp. 432–445.

[198] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems.” In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014, pp. 249–265.

[199] Y. Zhang, E. Ramadan, H. Mekky, and Z.-L. Zhang. “When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network.” In: *ACM Asia-Pacific Workshop on Networking*. ACM. 2017, pp. 1–7.

[200] Y. Zhao, L. Iannone, and M. Riguidel. “On the Performance of SDN Controllers: A Reality Check.” In: *IEEE Conference on Network Function Virtualization and Software Defined Network*. IEEE. 2015, pp. 79–85.

[201] Y. Zhou and J. Davis. “Open Source Software Reliability Model: an Empirical Approach.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 4. ACM. 2005, pp. 1–6.

[202] S. Zoppi, A. Van Bemten, H. M. Gürsu, M. Vilgelm, J. Guck, and W. Kellerer. “Achieving Hybrid Wired/Wireless Industrial Networks With WDetServ: Reliability-Based Scheduling for Delay Guarantees.” In: *IEEE Transactions on Industrial Informatics* 14.5 (2018), pp. 2307–2319.

List of Figures

List of Tables