MASTERTHESIS
Michael Müller

Availability Analysis of the ONOS Architecture

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Michael Müller

Verfügbarkeitsanalyse der ONOS Architektur

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Köhler-Bußmeier
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 28. February 2021

Michael Müller

Thema der Arbeit

Verfügbarkeitsanalyse der ONOS Architektur

Stichworte

SDN, ONOS, Consensus, Raft, Zuverlässigkeit, Verfügbarkeit, SPN

Kurzzusammenfassung

In dieser Arbeit vergleichen wir zwei ONOS Architekturen, die alte (vor v1.14) und die neue (ab v1.14) in Bezug auf ihre Verfügbarkeit und, ob die neue Architektur eine höhere Verfügbarkeit als die alte bieten kann. ONOS ist ein weitverbreiteter und populärer SDN Controller dessen Architektur mit Version 1.14 geändert wurde, um Software Upgrades während der Laufzeit zu ermöglichen. Um diese Frage zu beantworten, erstellen wir ein GSPN Modell, anhand dessen wir zeigen, dass die neue Architektur eine höhere Verfügbarkeit aufweist, vor allem wenn unzuverlässigere Hardware genutzt wird.

Michael Müller

Title of Thesis

Availability Analysis of the ONOS Architecture

Keywords

SDN, ONOS, Consensus, Raft, Reliability, Availability, SPN

Abstract

In this work, we compare two ONOS architectures, old (before v1.14) and new (v1.14 and after), in terms of their availability and answer the question if the new outperforms the old architecture. ONOS is a widely used and popular open source SDN controller that changed his architecture with version 1.14 to enable in service software upgrades. For this we create a GSPN model, upon which we can present that the new architecture has a higher availability, especially in environments with less available hardware.

Contents

List of Figures **vii**

List of Tables **ix**

Listings **x**

1 Introduction **1**

2 Basics **2**

2.1 Dependability 2

2.1.1 Reliability Attribute 2

2.1.2 Availability Attribute 3

2.1.3 Safety Attribute 3

2.1.4 Maintainability Attribute 3

2.1.5 Threats 3

2.1.6 Fault Tolerance 5

2.2 Consensus 5

2.3 Software Defined Networks (SDN) 5

2.4 ONOS 7

2.5 Stochastic Petri Nets (SPN) 7

3 Related Work **9**

3.1 Performance Evaluations with similar context 9

3.1.1 Reliability Prediction for Fault-Tolerant Software Architectures . . 9

3.1.2 Availability Modelling of Software-Defined Backbone Networks . . 10

3.1.3 Availability Modeling and Analysis for Software Defined Networks 11

3.1.4 Response Time and Availability Study of Raft Consensus in Distributed SDN Control Plane 12

3.1.5 Better Safe than Sorry: Modeling Reliability and Security in Replicated SDN Controllers 13

3.2 Performance Evaluations with similar tooling or formalism 13
3.3 Chapter conclusion 14

4 Model 15
4.1 The ONOS Architectures 15
4.2 Model Elements 16
4.2.1 Included model elements 17
4.2.2 Excluded model elements 19
4.3 Used Parameters 20
4.4 GSPN model in GreatSPN 23
4.4.1 Model explanation 23
4.4.2 Move to SWNs 36
4.5 Chapter Conclusion 36

5 Validation 38
5.1 Model Realization 38
5.2 Manual Simulation 40
5.3 Automated Simulations 41
5.4 Sensitivity Analysis 42
5.5 Simulation Results Plausibility 44
5.5.1 Basic Architecture Results 44
5.5.2 Failure Rates 46
5.5.3 Recovery Rates 46
5.5.4 Upgrade Rates 47
5.6 Chapter Conclusion 49

6 Evaluation 50
6.1 Sensitivity Results interesting for architecture comparison 50
6.1.1 Link fail and recovery rate 50
6.1.2 Hardware fail and recovery rate 52
6.1.3 Atomix software recovery rate 54
6.2 Scenarios 54
6.2.1 Upgrade scenarios 54
6.2.2 Different Atomix and ONOS counts 56
6.3 Evaluation Summary 58

7 Conclusion 60

Bibliography **62**

A Appendix **71**

A.1 Stacked Failure Extension 71
A.2 Simulation Tool 71
A.3 URL Links 74
A.4 ONOS Software failure rates to surpass 'no upgrade' scenario 75
A.5 Full GSPN model 75

Selbstständigkeitserklärung **77**

List of Figures

2.1 Logic placement in traditional and software defined networks, from [51] . . 6

4.1 Cluster of ONOS before v1.14 (left) and (after) v1.14 (right) 16
4.2 The white ONOS square in GreatSPN 24
4.3 The white ONOS cluster square in GreatSPN 24
4.4 The grey Atomix square in GreatSPN 26
4.5 The blue Partition square for ONOS instances in GreatSPN 28
4.6 The blue Partition square for Atomix instances in GreatSPN 29
4.7 The green upgrade (v1 to v2) square in GreatSPN 32
4.8 The pink cluster metadata square in GreatSPN 34
4.9 The used parameters and their default values in GreatSPN 35

5.1 Cluster availability (blue) and linear trend line (black) of the $ONOSv2SWFailRate$ parameter steps in the new architecture 45
5.2 Cluster availability (blue) and linear trend line (black) of the $ONOSv3SWFailRate$ parameter steps in the new architecture 45
5.3 Cluster availability (green) and linear trend line (black) of the $AtomixSWFail$ parameter steps in the new architecture 46
5.4 Cluster availability (blue) and linear trend line (black) of the $ONOSSWRecRate$ parameter steps in the new architecture 47
5.5 Cluster availability (green) and linear trend line (black) of the $UpgradeClusterRate$ parameter steps in the new architecture 48
5.6 Cluster availability (green) and linear trend line (black) of the $ONOSUpgradeFailFinishedRate$ parameter steps in the new architecture 48

6.1 Cluster availability (green, blue) and linear trend line (black) of the $LinkFailRate$ parameter steps in the new (left) and old (right) architecture 51
6.2 Cluster availability (green, blue) and linear trend line (black) of the $LinkRecRate$ parameter steps in the new (left) and old (right) architecture 51

6.3 Cluster Availability (blue) and linear trend line (black) of the *HWFailRate* parameter steps in the new (left) and old (right) architecture 52
6.4 Cluster Availability (blue) and linear trend line (black) of the *HWRecRate* parameter steps in the new (left) and old (right) architecture 53
6.5 Cluster availability (blue) and linear trend line (black) of the *AtomixSWRecRate* parameter steps in the new (left) and old (right) architecture 53
6.6 Cluster availability of the old and new architectures, including different upgrade behaviours 55
6.7 Extension of Figure 6.6, Cluster availability of the new architectures, including different upgrade behaviours 56
6.8 Cluster availability of the recommended deployed instance counts 58

A.1 Basic idea for stacked failure extension 72
A.2 Our model in GSPN and GreatSPN 76

List of Tables

4.1 Bugs per hour from ONOS v1.10 to v1.14 22
4.2 Used failure, recovery and upgrade rates 22

6.1 Decreases of the ONOS software failure rates in the 'increase' scenarios . . 55

A.1 Decreases of the ONOS software failure rates to surpass the 'no upgrade' scenario 75

Listings

A.1 Example simulation tool configuration 73
A.2 Full URLs of shortened links 74

1 Introduction

In 2014 the development of software defined networks (SDN) rapidly gained traction which kept up till today. Since then, ideas have been implemented, tested and altered. [5, 13, 16, 67, 61, 19, 30, 54, 65, 57, 20, 12, 6]. One such alteration was the extraction of the embedded data store in ONOS into a separate executable, how this change affected ONOS' overall availability is the focus of this work.

ONOS is one of the most popular open source SDN controllers, and it uses Atomix as its shared data store for a logical centered but physical distributed deployment. Atomix implements the Raft consensus protocol, for which the number and location of Atomix instances need to be immutable. From ONOS version 1.14 onwards, ONOS and Atomix are separated, this separation allows ONOS to be upgraded and horizontally scaled much easier than before. Performance analysis of architectures [36, 26, 14, 28, 62, 42, 35, 33, 34, 53], SDNs [51, 52, 54, 40] and older ONOS versions [1, 2, 4] have been done before, but not with the newer ONOS versions and not with focus on its architectures. Additionally, authors write about the need for more performance evaluations in SDNs, especially regarding reliability and availability [45, 66, 52, 51, 30, 20].

As ONOS has availability within its core focus, this raises the question, if this architectural change was beneficial to ONOS' availability. To the best of our knowledge, this is the first attempt to evaluate and compare the availability of the two ONOS architectures. The contribution of this master thesis is an availability analysis to determine under which circumstances which architecture, old or new, offers higher availability. This analysis is based upon simulation results of a generalized stochastic Petri net model simulated within the tool GreatSPN.

The master thesis is structured as follows: In Chapter 2, we describe important basics for this work. Published work that is related to this thesis, either by a similar goal, tool or formalism, is presented in Chapter 3. Then, we present and validate our model in Chapter 4 and Chapter 5 respectively. The last two Chapters 6 and 7 answer the research question, conclude the paper and point out interesting future work.

2 Basics

In this chapter relevant basics are presented and explained, so the reader is well suited for the upcoming technicalities. First we will describe dependability and its elements in Section 2.1, of which 'availability' is of special interest. Then we will continue with software defined networks in Section 2.3, which includes an introduction to the ONOS application. As ONOS makes use of a consensus protocol, we introduce general aspects of it in Section 2.2. At last, we present stochastic Petri nets in Section 2.5, which form the basis for our analysis.

2.1 Dependability

Dependability is a group of concepts and attributes, each is explained in this section. [8, 60]

2.1.1 Reliability Attribute

Reliability is the continuity of correct service [8]. A highly reliable system is a system that continuously works as expected for a long period of time, at best forever. An important metric for this attribute is 'Mean Time To Failure' (MTTF), as the name suggests, this represents the average of how long the service works without interruptions. The measurement of how long it takes to repair the interruption is 'Mean Time To Repair' (MTTR). [29, 60]

2.1.2 Availability Attribute

Availability is the probability for correct service in a given moment. A highly available system is a system that has a high probability to work as expected in any given moment, at best always. This probability can be calculated with $\frac{MTTF}{MTTF+MTTR}$. [8, 60, 29]

Once the system is stabilized and the availability is roughly a constant value, we can talk of it as being the 'Steady State Availability'. This is widely used in previous publications as the basis of their availability evaluations. [42, 53, 36, 35, 27, 24, 62, 8, 44, 54, 23, 45]

2.1.3 Safety Attribute

Safety means the '[...] absence of catastrophic consequences on the user(s) and the environment' [8]. [60] gives the example of control systems in nuclear power plants or spacecrafts, if '[...] such control systems temporarily fail for only a very brief moment, the effects could be disastrous'.

2.1.4 Maintainability Attribute

This attribute addresses how easy it is to repair failed systems. Automatic and / or easy maintenance can help to improve the availability of a system, this for example can be observed when the maintenance needs less time which leads to a lower MTTR and to a higher availability. Another example is preemptive maintenance in form of an upgrade that fixes bugs to increase reliability of a system. [8, 60]

2.1.5 Threats

Threats to the dependability of a system are faults, errors and failures.

Faults are the basis of the threats, they can activate errors. They are either introduced during development, via incorrect code, due to physical problems, e.g. an old hard drive stops working, or are generated from outside the system, e.g. during interaction with the user. [8]

Errors are part of the system's state and may activate failures if the error has external consequences [8, 60]. Errors can be detected if they generate any kind of message or signal [8]. Regarding their timing they can be [60]:

Temporary: Will occur once, this error may be resolved by repeating the process.

Permanent: Is equal to an unrecoverable disaster, this can only be repaired by exchanging the fault part.

Reoccurring: Will occur and resolve multiple times, e.g. a loose connection in an electronic circuit.

[8] defines **failures** as '[...] an event that occurs when the delivered service deviates from correct service', they can lead to the activation of further faults. [60] describes failures as unmeet specifications. An example failure is an *uncaught error* of a software that leads to a complete crash of the software. This crash failure example is one of multiple failure models, in the following a selection [8, 60, 38, 9, 15]:

Crash Failure: A system is fully working until its crash, if the crash is permanent its also called a 'crash-stop' failure mode. More examples are 'crash-recovery' for eventually recovering systems or 'crash-safe' for failures that do not harm the system.

Omission Failure: A system does not send any messages. This can be the result of multiple errors.

Timing Failure: The response time of a system is outside the specified interval. For example this can be fixed with offloading load, in this case the error will reoccur until the system load is within a tolerable range or the clients adjust to higher response times.

Response Failure: A response is in the wrong format or its values are incorrect. For example this can be due to faulty code, in this case the error is permanent or reoccurring.

Byzantine Failure: The failure may cause the system to fail '[...] in any possible way [...]' and '[...] may be unobservable [...]' [60]

2.1.6 Fault Tolerance

[60] calls a system fault tolerant if '[...] it can continue to operate in the presence of failures'. This includes the detection and right response upon detection. In other words, fault tolerance helps to avoid failures through error detection and an automatic maintenance [8, 14]. In [9, 8] fault tolerance is seen as the basis for dependability.

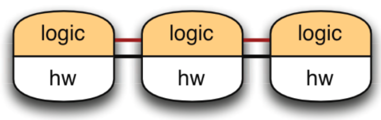
2.2 Consensus

Consensus protocols help to synchronize a state across distributed systems. In consensus protocols there is one leader which will order incoming updates. It will then ask the participating systems if a certain order is consistent with their respective state. If the majority of systems acknowledge the update, it is committed by the leader and each system applies the update. The same way consensus can be used to elect a participating system to their leader. [70, 39, 38, 54, 60, 64]
One such consensus protocol is Raft[1]. Raft can work if the majority of the participating instances are available. To be specific, the majority is defined as $1 + \frac{instancecount}{2}$. [70, 5, 67, 39, 32, 38, 54]

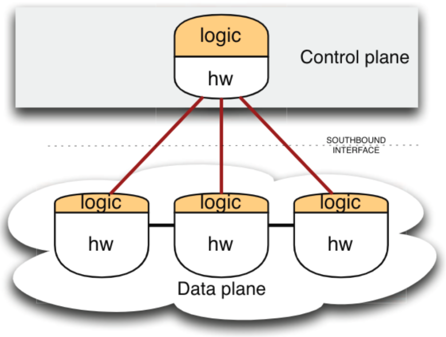
2.3 Software Defined Networks (SDN)

Networks have a control and a data plane, while the control plane is more of a logical nature, the data plane is more of a physical nature. The control plane can insert, modify or delete forwarding rules to impact routes through the network. The data plane forwards packets from one ingress port to certain egress ports according to the set forwarding rules. In traditional networks, both planes reside in each router. SDNs split these planes, the SDN controller has the control over the routers which are only left with the data plane. Routers are then called SDN switches.
The control plane can be 'in-band', on the same links as the data plane, or 'out-of-band', on own links. As an example, when in an out-of-band control plane a controller has a direct connection to a certain switch, the same connection in in-band control planes may require additional switches in between. [58, 63, 21, 16, 51, 52]

[1]Raft Visualizations: http://thesecretlivesofdata.com/raft/, https://raft.github.io/



(a) Current IP networks: Distributed logic embedded in the forwarding engine of network nodes.



(b) SDN: Logically centralized control logic combined with simplified network elements.

Figure 2.1: Logic placement in traditional and software defined networks, from [51]

The controller communicates in three major ways [58, 63, 21, 13, 16, 51, 52]:

- ’Southbound’ (e.g. via OpenFlow) to the switches: As long as they have a common southbound protocol, SDN controller can communicate to software or hardware switches. Depending on the southbound protocol in use, the switch has one or more masters. A switch may report changes in the network (e.g. link is down) only to its master.
- ’Northbound’ (e.g. via HTTP) to SDN applications: SDN applications have the possibility to modify the network according to their needs, this flexibility is often mentioned as one of the main reasons to adopt SDNs. For example, if a latency sensitive application needs a low latency connection from one of its users to a certain server, it can request the SDN controller to create such a route. The controller will then make sure the route is created according the specified requirements, in this example the low latency.
- ’East-/Westbound’ to other SDN controllers: The inter-controller communication is for example used to share the network state. Multiple controllers can be used to balance the load or to increase the availability of the control plane. The possibility to only use cheaper off the shelf hardware is an important reason for the adoption of SDNs. This also enables further programmatic verification and debugging of networks.

SDNs have the benefit, that protocols and devices are easier to update or replace. Furthermore, SDNs can improve resource optimization, ease of maintenance and ease of operation. [51, 52]

2.4 ONOS

The 'Open Network Operating System' (ONOS) is, next to OpenDaylight, the largest open source SDN controller [67]. It is widely mentioned in publications, for example [61, 31, 59, 57, 54]. ONOS was created in 2014 on the basis of the Floodlight SDN controller. Its target was to tackle high throughput, low latency, large network state sizes and high availability, detailed performance numbers can be found in [13]. To handle this in an efficient and partition tolerant manner, the controller is physically distributed and logically centered. It uses the Raft consensus algorithm to synchronize the shared network state between instances, currently implemented via Atomix [39, 13, 5, 65].

2.5 Stochastic Petri Nets (SPN)

We describe in what follows an extension that adds time into Petri nets, the so called stochastic Petri nets (SPN). In our work we create SPNs with the tool GreatSPN.

SPNs are Petri Nets with the extension that transitions have a timer which makes them 'timed transitions'. If enough tokens are provided in the respective input places, the timer of a transaction runs, when a timer runs out it fires. Timers with zero time are called 'instantaneous', if we have both timed and instantaneous transitions in an SPN, we have a Generalized SPN (GSPN). If an instantaneous transition is ready to fire, it will do so before any other timed transition. [37, 49, 48, 10]

When a transition is disabled, its timer stops running. When the transition is enabled again, there are two main approaches. First, to start the timer from where it left ('age memory') or second, reset the timer ('enabling memory'). Furthermore, the probability that two timers run out at the same time is zero. [48]

Besides additional transitions, a GSPN also add 'inhibitor' arcs. They are the exact opposite of what an input arc is. If a place contains a specified amount of tokens ore more, the transition connected via an inhibitor arc is disabled. [48]

A (G)SPN can be converted to a Markov chain for evaluation [37, 49]. This evaluation can be used to assess characteristics like performance, reliability, fault recovery, fault tolerance and fault coverage [37, 49]. The chosen tool, GreatSPN, is able to create and evaluate GSPNs via the mentioned conversion to Markov chains. It is also well-established for this use case. [10, 47, 17, 7, 22]
To avoid confusion, we will always write out 'GreatSPN' and never abbreviate it with 'GSPN'.

3 Related Work

This chapter will present published work with a similar context in Section 3.1 and work with similar tools or formalism in Section 3.2.

3.1 Performance Evaluations with similar context

In this section we present published work with a similar context.

3.1.1 Reliability Prediction for Fault-Tolerant Software Architectures

[14] use reliability prediction in the context of Software Product Lines (SPL) with optional fault tolerance mechanisms on different levels. An SPL consists of a core and adjustable extensions, which can lead to multiple versions of the same basic software. They create a tool to analyze all variations of a given SPL. The model and their reliability solver is based upon the Palladio Component Model and implemented via the Eclipse Modeling Framework. Their tool converts the model into Markov-based prediction models for evaluation.

Their model considers the reliability of the software, hardware and network layer. Each layer has its own basis for the reliability prediction:

- On the **software** layer they have an action's probability to fail and the probability of the (nested) recovery blocks to fail.
- On the **hardware** layer they use steady state availability, this is based upon the MTTF and MTTR of a given resource like CPU or disk. Resources can either be 'OK' or 'FAIL'. Hardware failures are expected to be independent of each other.

- On the **network** layer a coarse grained link availability is considered, with the expectation that every communication has a request and a response that can fail. Single links are expected to be independent of each other.

For evaluation, they test their tool in two case studies regarding its sensitivity, by changing certain parameters, and accuracy, by comparing the numerical approach with multiple simulations that ran 10^7 and 10^6 seconds for the first and second case study respectively. In the first case study they create an SPL with artificial failure probabilities and in the second they extend previously published work by considering e.g. new fault tolerance mechanisms.

3.1.2 Availability Modelling of Software-Defined Backbone Networks

The authors of [51] predict the steady state availability of SDNs and traditional IP backbone networks. Unclear is if they used an in-band or out-of-band control network.

Their model considers the availability of traditional IP routers, SDN switches, SDN controllers and links. The model is split in two hierarchical layers to avoid a potential '[...] uncontrolled growth in model size [...]'. First the **structural** model, it sees the network elements either as independent of each other or as sets of network elements that may experience multiple errors with a vaguely described '[...] advanced recovery strategy [...]'. It is modeled with structure functions based on minimal cut and path sets. Second the **dynamic** model that focuses on the dependencies between the network elements and their multiple errors. This is modeled with a Markov model. A Network is up if all hosts can reach all other hosts and if all hosts can reach at least one SDN controller. Each network element has its own availability relevant metrics, see their Table 2, and basis for the availability prediction. While the links are solely dependent on their hardware, all other elements also consider their respective software availability.

- **Traditional IP routers** are assumed to contain one primary and one backup controller. More detailed failure scenarios are omitted as they are infrequent, not very probable and do not have a significant impact on the results.
- The **SDN switch** are without operation and maintenance errors due to their simple design in comparison, the idea is that its failure rate is pretty low and even then the reparation should not take much time.

- For the **SDN controllers** it is considered that these are operated upon processors. Only when there are less working processors than the SDN controller would need, it fails. Furthermore, the authors consider software, operation, maintenance and coverage failures. Due to the importance of SDN controllers in SDNs, their impact if they fail is increased by the number of network elements. The basic assumption is that both traditional and software defined networks need the same processing power in total, in SDNs this power is needed for the SDN controllers in contrast to the distribution in traditional networks.

For evaluation, they perform sensitivity tests with two case studies, on a 'national' and 'world-wide' backbone network. First contains four 'sites', ten switches and two SDN controllers. The second contains ten 'sites', 28 switches and two SDN controllers.

3.1.3 Availability Modeling and Analysis for Software Defined Networks

The context in [52] is a generic SDN. The author's goal was to assess the availability of an SDN using a stochastic availability model to calculate the steady state availability and analyze the downtime.

They propose a hierarchical availability model that prevents the '[...] state-space explosion problem [...]' that might occur with a large monolithic approach. The upper layer contains reliability graphs and focus on the network. It captures how the hosts are connected and represents the capability of an SDN to form multiple network topologies. The lower layer focuses on devices and uses an approach that is based upon SPNs, stochastic reward nets (SRN). Each device has its own SRN. Finally, SDN controllers are able to create links between two devices. Failures of the SDN controller are not considered. The model was implemented using the Symbolic Hierarchical Automated Reliability and Performance Evaluator.

Each device has its own basis for the availability prediction in the SRNs:

- **Virtual machines, hosts and links** are based upon their MTTF and MTTRec. These failures are propagated, e.g. if a virtual machine fails all services on it fail too.

- The **SDN switch** can have downtime due to a number of reasons: Upgrades, failure of chassis hosting components (e.g. power supply), network interfaces (ingress and egress), internal cards, the gigabit routing processor and the '[...] internetworking operating system [...]'.

- For evaluation, they have a **storage** device in the network for which they consider failures and repairs of single disks in a manual process.

Their evaluation contains an artificial SDN with six hosts, three switches, one controller and one storage. The evaluation considers a certain network communication between certain hosts via certain routes. Only if these hosts can continuously have a successful communication, the SDN is considered up. In their Table 2 they show their input in the evaluations, these include e.g. a VMs MTTF, MTTR, time to boot and also mean time to upgrade a switch. They conclude that link availability is of vital importance for SDN steady state availability.

3.1.4 Response Time and Availability Study of Raft Consensus in Distributed SDN Control Plane

The goal of [54] is to evaluate the response time and availability of distributed SDN clusters regarding the old (pre v1.14) ONOS architecture and ODL with Raft as its distributed data store.

They use stochastic activity networks as a model generation framework, which they prefer over GSPNs due to two points. Its additional inhibition of transitions and the '[...] flexible predicate assignment to the gate abstractions [...]'. One of their three models is interesting for our work. The 'RAFT Recovery SAN Model' which contains hardware and software (Java OSGI, Processes) failures that can impact the SDN controllers. All failures follow a negative exponential distribution. To evaluate the response time they also model failure injection to cause further failures that can be correlated. They consider an in-band control network with a coarse-grained static data plane reliability.

For evaluation, they compare their models to an experimental Raft setup with no failures and to an ODL setup. The failure probabilities are gathered from the ODL controller and published work. They also transform their models to Markov chains via the Möbius tool. One result is that if five or more controllers are in the cluster, the steady state availability is higher than 99,987659%. In their evaluation, they also analyze the complexity of their Markov chains and propose the evaluation of the worst case only, to be able to scale out this performance evaluation approach for larger models.

3.1.5 Better Safe than Sorry: Modeling Reliability and Security in Replicated SDN Controllers

The goal of [40] is to offer mitigations on the basis of their proposed modelling framework that considers reliability, availability and security in distributed consensus protocols like Raft.

They use Figaro as a language to create a knowledge base from which they generate their model. Their model considers among others, failure probabilities from published work (includes ONOS related ones), detectability of failures and different repair rates. They do not consider downtime due to attacks.

Their evaluation is based upon Markov chains generated from the model above and explored via the Monte Carlo simulation via the YAMS tool. With three controllers the availability of Raft is at 70%, with seven it is at 99,9%. With 4, 7 or 10 controllers, the addition of one or two controllers decreased the overall availability.

3.2 Performance Evaluations with similar tooling or formalism

In Chapter 2 we presented the GreatSPN tool and the GSPN formalism. Both are widely used in the context of performance evaluation, for example in these publications:

- **Validation and evaluation of a software solution for fault tolerant distributed synchronization [11]**: Presents a case study to analyze and evaluate UML diagram types with the use of multiple tools. GreatSPN and Stochastic Well-Formed Nets (SWN, which are coloured GSPNs) are used for translation into another tool with the 'GreatSPN-to-PROD' utility, as a solver with the 'algebra','Multisolve' utilities and to create images of the models. They choose GreatSPN for these tasks for its efficient solution methods regarding the analytic approach and simulation. They describe similarities between GSPN and SWN in regard to GreatSPN, but they do not explain why they choose SWN over GSPN. As they make heavily use of the colours in the SWN, we assume that this is the reason.

- **Performance Modelling for the CSMA/CD Protocol Using GSPN [41]**: Evaluate performance characteristics of a LAN with a single bus and multiple devices. GreatSPN and GSPN are used for validation and solving.
- **Distributed transactions on mobile systems: performance evaluation using SWN [55]**: Validation of a transaction protocol in a mobile environment. GreatSPN and SWNs are used for validation and solving with the 'WNSIM' utility.
- **Freshness-Aware Metadata Management: Performance Evaluation with SWN models [22]**: Analyzes the performance of query routing to create a new algorithm for needed metadata. GreatSPN and SWNs are used for validation, evaluation and simulation with the 'WNSIM utility'. They point out that GreatSPN is the only tool that permits symbolic simulation with SWNs, which is heavily used in their work.
- **On the use of formal models in Software Performance Evaluation [46]**: Creates a translator from UML activity diagrams to GSPNs for performance evaluations, which extends their previous work with similar goals. The created GSPNs are recommended to be analyzed with GreatSPN.

Interested readers are directed to the following papers for more examples: [56, 68, 43, 18, 25, 69, 26, 71, 28]

3.3 Chapter conclusion

Now with the relevant basics and related work in mind, we have a foundation on which we can begin to formulate our research question and create & validate our model in the next chapters.

4 Model

In this chapter, we will introduce our assumed model. We will present the analyzed ONOS architectures and define our research question in Section 4.1. After that, we will explain what the model includes in Section 4.2.1 and what it excludes in Section 4.2.2. At last, Section 4.3 presents the used parameters and Section 4.4 the final GSPN model in GreatSPN.

4.1 The ONOS Architectures

The work on a new ONOS architecture arouse 2017 with the formation of the ONOS internal 'In Service Software Upgrade' team[1]. An ISSU means to upgrade software that is currently running and answering requests. The goal is to upgrade without a loss of availability. For this goal the ISSU team decided to change ONOS' architecture: '[...] In past versions, ONOS embedded Atomix nodes to form Raft clusters, replicate state, and coordinate state changes. In ONOS 1.14, that functionality is moved into a separate Atomix cluster'. [5]

The differences between the architectures is best presented with two example clusters. The example cluster of an old ONOS architecture in Figure 4.1 contains three item types, ONOS controllers (C1-C3, squares), SDN switches (S1-S3, circles) and hosts (H1-H3, hexagons). Each controller has contact to each switch (green dotted lines), but only one controller is the master of a switch (bold green dotted lines). Also, there is a communication between the controllers (purple dotted lines). The black lines between switches and hosts are signaling the connectivity between these items. The example cluster of a new ONOS architecture in Figure 4.1 is noticeably noisier. Here we have one item type more in the cluster, Atomix instances (A1-A3, triangles). Additional to the Atomix instances, we have more links, these connect each Atomix instance with each

[1]Also called ISSU Brigade lead by Jordan Halterman

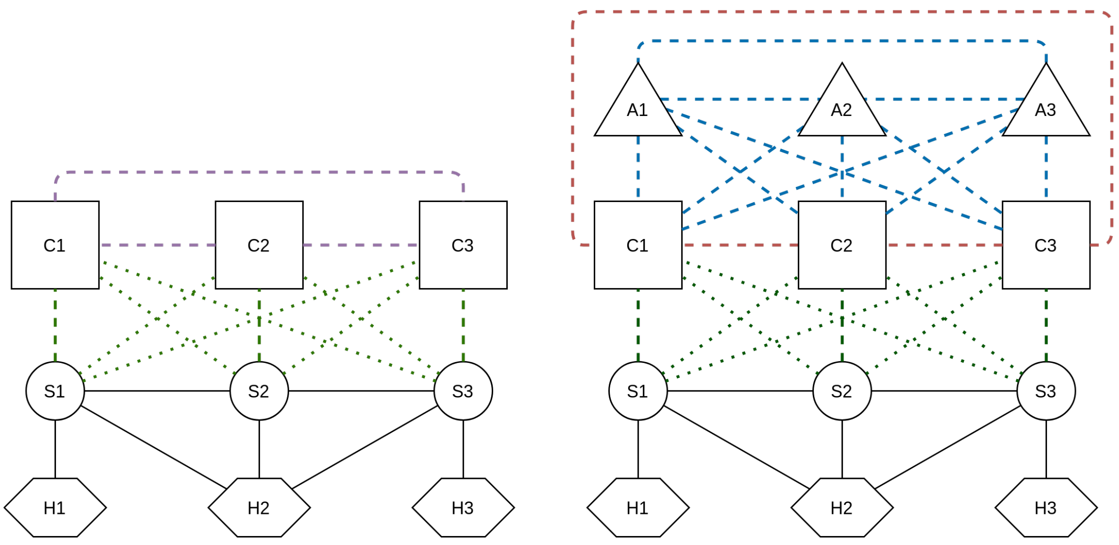


Figure 4.1: Cluster of ONOS before v1.14 (left) and (after) v1.14 (right)

controller (blue dotted lines). Our controllers still have their controller to controller communication (red dotted lines).

The interesting point is the mentioned additional noise in the new example cluster. This separation of Atomix and ONOS brings certain flexibility benefits like dynamic horizontal scaling of the ONOS instances and separate & easier upgrade of ONOS and Atomix instances. It also brings some costs in form of additional links, communication, instances and nodes, if as shown in the example clusters each instance is deployed on its own node. This leads us to question if this new architecture benefits the overall availability. We want to analyze the following points:

- Does the additional complexity in the cluster, as described above, harm the overall availability?
- If so, does it only decrease cluster availability in certain scenarios?

4.2 Model Elements

To make the description of the model easier, we classify elements of our model as either objects or behaviour of these objects. In the following we will present included and excluded model elements and argue why.

Objects are either *up* or *failed*, like for example in [14]. *up* objects can fail, *failed* objects can recover. Initially all objects are *up*.

4.2.1 Included model elements

The included elements are enumerated with the prefix *EO* for objects and *EB* for behaviours.

EO1 We need to consider **ONOS instances** in our work, including four versions of ONOS. Each ONOS version has its own software availability. We consider four ONOS versions as ONOS releases quarterly and with four versions we represent one year.

EO2 We also consider **Atomix instances**. Together with the ONOS instances, these elements are the core of our model.

EO3 We consider the **control plane links**, between Atomix & Atomix and between ONOS & Atomix.

EO4 As an own object we model the **consensus protocol status**. The consensus protocol is *up*, if the majority of Atomix instances are *up*. Otherwise, it is *failed*.
Reason: This is due to the basic requirement of the Raft consensus protocol, which needs an active participation of the majority of instances as explained in Chapter 2. In the following sections and chapters we will use 'consensus' and 'consensus protocol' interchangeably.

EO5 The **cluster availability** is also modeled as an object. It is *up*, if consensus and at least one ONOS instance is *up*, in any other case it is *failed*.
Reason: Only if consensus is *up*, which includes that enough Atomix instances are *up*, and at least one ONOS instance is *up*, the cluster can work as intended and so only then it is available.

EB1 **Failure of ONOS and Atomix** instances can be due to hardware failures, software failures and network partitions. Every instance has its own hardware node and all nodes are equal.
Reason: The consideration of these failures is intuitive. Besides that, we assume that each instance has its own node to achieve a simpler model in contrast when we would consider shared nodes. This node separation can also be found in [51].

EB2 If consensus fails, the **Atomix and ONOS instances become idle** and can no longer fail from software problems.
Reason: This is based upon the behaviour of Atomix and ONOS described in [5], in that case they can not fully answer incoming requests and only execute a limited amount of their code, which again means less possibility to execute faulty code.

EB3 For the old ONOS architecture we consider the **combined failure of ONOS and Atomix**. If one fails, the other fails too.
Reason: This is due to the deeply coupled deployment of ONOS and Atomix instances in the old architecture as described in [5].

EB4 We consider worst case **network partitions** by accounting the number of failed links. Once a certain amount of links has failed, one respective instance fails. Links are either between the Atomix instances or between Atomix and ONOS instances. In the example in Figure 4.1 this would mean that for each two Atomix links one Atomix instance fails, or that after three failed ONOS-Atomix links one ONOS instance fails.
Reason: The consideration of link failures and partition is an interesting topic for this work as it hits one of the main points of the architecture change, the additional links. We do not consider specific node connections of links, as it would drastically increase the models complexity and also would overstep set time limits of this work. As for why we see this as a failure, with reference to [5]: Atomix / ONOS instances stop serving requests and wait for the partition recovery once they are partitioned. This is similar to the case when consensus fails, as explained above. The ONOS-Atomix partition does not lead to a failed Atomix instance, as Atomix is not dependent on ONOS instances, but it is the other way around.

EB5 In the new architecture, **the cluster can be upgraded** if all Atomix and ONOS instances are *up*. First Atomix will be upgraded, to be compatible with the new ONOS version, then ONOS is upgraded. The influence of the upgrades onto ONOS' availability are dependent on the used parameters for each version and theoretically could improve or worsen its availability. We will perform a rolling upgrade, upgrading one instance after another. This also means that the multiple versions are compatible to each other. The upgrade can fail, which leads to a longer upgrade time, but it can not fail entirely, so a rollback is not necessary to consider.
Reason: The rolling upgrade procedure described is based upon their mailing list[2]

[2]See https://tinyurl.com/u39fj9px, https://tinyurl.com/2n8u4jfp and https://tinyurl.com/2yb86t8b

and [5]. That an upgrade only temporary fails, is based upon the assumption that an expert performs the upgrade and also verifies the success, as the ISSU team mentions in their presentation[3] of the upgrade procedure.

4.2.2 Excluded model elements

EO6 Our model focuses on the control plane, so the **data plane** is not considered.
Reason: As it can be observed from the old cluster to the new cluster in Figure 4.1 the connections (green dotted lines) between SDN controller, SDN switches and hosts do not differ. We reason that the architectural change in the control plane does not affect the data plane in any way and so the impact of the data plane on the overall availability is equally in both architectures and so can be omitted as we are only interested in the changes between the two architectures.

EB6 We model a **failure detection** probability of 100%, which in other words excludes it from the model.
Reason: This approach simplifies our model and can also be found in [51, 52].

EB7 We do not consider **load** related software, hardware or link failures.
Reason: In our opinion, this would lead to a much more complex model that would cost sparse time for creation we rather spend on other areas we think have a greater benefit for this work.

EB8 The **stacking of failures** is not considered. For example on top of being already partitioned, one ONOS instance fails due to a software failure.
Reason: An idea of how complex it is to model this detail, can be seen in the Section A.1 in the Appendix. It would extend the time limits of this work to incorporate this into the model.

EB9 As described in Section 4.2.1, **Atomix** instances must be **upgraded** before ONOS can, but this has **no impact on the overall availability**.
Reason: Since this upgrade is done so Atomix is compatible with the new ONOS version, we do not expect it to also include an impact of availability for example through bug fixes. Henceforth, we exclude this detail from our model.

[3]See https://wiki.onosproject.org/display/ONOS/ISSU within [5]

EB10 The **old architecture can not be upgraded**.
Reason: We reason this exclusion with our goal to compare between less complexity and less flexibility in the old architecture, against more complexity and more flexibility in the new architecture, as discussed at the beginning of this chapter in Section 4.1.

EB11 The **gossip protocol** that is implemented in ONOS is excluded.
Reason: As the gossiping is only used for bidirectional exchange of state, it is used to detect smaller state drifts between instances and to bring new instances faster up-to-date. If consensus is not longer working as mentioned above, the gossip protocol can't work either. [5]

EB12 Unavailability due to attacks or other **security** related threats to availability are not modeled.
Reason: As security is a whole new topic that has a depth on its own which would extend the scope of this work too much. This approach can also be found in [40], as mentioned in Chapter 3.

4.3 Used Parameters

According to the included elements explained above, we will now present the used parameters. As for wording we use 'parameter' and 'rate' interchangeably.

The ONOS parameters sourced from [40, 50, 4] do not include Atomix. We can therefore use them as purely ONOS parameters. This is important to point out as in older versions Atomix and ONOS were very closely coupled, as described in Section 4.1.

For Atomix we could not find any published work that states specific numbers for it. What we could find was a software failure rate of one week in [54], but this value is for the complete SDN controller that implements the Raft consensus protocol. The same programmers that wrote ONOS wrote Atomix. Therefore, we assume that both have the same reliability, so we set the **Atomix failure rate** equal to the failure rate of ONOS v1.13.

The Atomix software repair duration is set by us to 10 seconds. This number is gathered by a coarse grained test we did by starting and stopping an Atomix container within a Virtual Machine with 6 vCPUs and 14 GB RAM with an i7-9750H. We assume that an ONOS deployment could be done via Kubernetes, as they suggest on their website[4]. The number above is reasoned like following, one second is the time Kubernetes needs to perform liveness and readiness probes to check the status of the Atomix instance. If the instance is detected as unhealthy, it is shutdown which takes up around two seconds and starts up a new one on the same host which takes up roughly seven seconds. As we deploy the new Atomix instance on the same node we need to wait for the old one to be shutdown.

Getting the Atomix and ONOS upgrade durations is hard, as there exists no documentation for either one online. Furthermore, the tools that an installed ONOS version provides, do not contain hints on how to upgrade an existent cluster.
As the upgrade of Atomix and ONOS instances is mainly done via shutting down the old version and starting a new version instance in its place[5], one could argue that we plainly could take the time that needs. So for Atomix that would be nine seconds, as described above minus the one second for failure detection that is not necessary in this proactive process. For ONOS a restart takes around seven seconds, measured in the same environment. The problem is that these numbers seem pretty low for a major version upgrade. For this reason we set the upgrade duration arbitrarily.
For **Atomix upgrade** this means five minutes with regard to the easy upgrade as described in Section 4.2.2. We set the **ONOS upgrade** durations to 15 minutes, for the failure free upgrade, and four hours, when failures are encountered by the expert. 15 minutes, since we assume that upgrading ONOS is more complex than to upgrade Atomix due to expected data migration. The four hours are a mean of considering pretty simple errors, like wrong IP address in configuration, and more complex ones, like (partly) failed data migration.

In Table 4.1 we listed all sourced or calculated bugs per hour per ONOS version. Bugs per hour of v1.10, v1.12 and v1.13 are based upon published work, these are also the basis for our calculation of v1.11, v1.14 and the upgrades after v1.14. Version 1.14 and the upgrades are based upon the logarithmic trend, as we expect that fewer bugs are getting fixed for each release as it gets more time intensive to fix them.

[4]See https://tinyurl.com/fhkzmvae
[5]See https://wiki.onosproject.org/display/ONOS/ISSU within [5]

Version	Bugs per hour	Relative to v1.10	Source
v1.10	0,01750	100,00%	From published work: [40, 50]
v1.11	0,01517	86,67%	Calculated: mean of v1.10 and v1.12
v1.12	0,01080	61,71%	From published work: [4]
v1.13	0,01050	60,00%	From published work: [4]
v1.14	0,00885	50,56%	Calculated: Based upon logarithmic trend from Excel: $f(x) = -0,005386747003005 * ln(x) + 0,017517342986504$
1. Upgrade	0,00787	44,95%	
2. Upgrade	0,00704	40,20%	
3. Upgrade	0,00632	36,09%	

Table 4.1: Bugs per hour from ONOS v1.10 to v1.14

Rate and Source	Value
ONOS	
ONOS v1.10 failure rate (from [40, 50])	50 per year
ONOS v1.13 failure rate (calculated based upon v1.10 failure rate and the relative percental difference from v1.13 to v1.10 in Table 4.1)	30 per year
ONOS v1.14 failure rate (calculated like v1.13 failure rate)	28,28 per year
ONOS 1. Upgrade failure rate (calculated like v1.13 failure rate)	22,47 per year
ONOS 2. Upgrade failure rate (calculated like v1.13 failure rate)	20,10 per year
ONOS 3. Upgrade failure rate (calculated like v1.13 failure rate)	18,05 per year
ONOS v1.10 recovery rate (from [40, 50])	2 per hour
Atomix	
Atomix Software failure rate (see above, like ONOS v1.13)	30 per year
Atomix Software recovery rate (see above)	6 per minute
Upgrade	
Cluster upgrade rate (see Section 4.2.1 element **EB4**)	4 per year
Atomix upgrade rate (see above)	12 per hour
ONOS upgrade rate (see above)	4 per hour
ONOS upgrade rate with encountered failures (see above)	6 per day
Hardware	
Hardware failure rate (from [54, 51])	2 per year
Hardware recovery rate (from [54, 51])	2 per day
Link failure rate (from [51])	3 per year
Link recovery rate (from [51])	4 per hour

Table 4.2: Used failure, recovery and upgrade rates

4.4 GSPN model in GreatSPN

In the following sections we will present our GSPN model in detail and explain why we did not change our formalism to SWNs. Specific parts of the model will be shown in the following sections. The full model can be seen in Figure A.2 in the Appendix.

4.4.1 Model explanation

In the next sections we will describe the model in depth, this is structured according to the squares in which the model is visually split with different background colours:

White squares : ONOS model elements

Grey square : Atomix model elements

Blue squares : Partitioning of Atomix and ONOS instances

Green squares : Upgrading elements

Pink square : Cluster metadata, like cluster availability or which architecture (old/new) is represented

Orange square : Contains the used parameters and their default values

The name of the elements are adjusted to their type. Places should reflect the current status in its name, e.g. $ONOSAtomixLink_{up}$ or $ONOSAtomixLink_{failed}$. Transitions should reflect that something is happening, e.g. $PartitionONOS_{v1}$ or $LinkFails_{ONOSAtomix}$. Parameters have the suffix 'Rate', e.g. $LinkFailRate$ or $LinkRecRate$. Note that 'Rec' is an abbreviation of 'Recovery' to make the model easier and to use the shared space more efficiently.
In the following we will speak of 'versioned' elements, this means that this element exists with different versions, for example O_{up} with O_{upV1}, O_{upV2} or $PartitionRecovers_{ONOSAtomix}$ with $PartitionRecovers_{ONOSv1Atomix}$ and $PartitionRecovers_{ONOSv2Atomix}$.
We also use the terms 'ONOS cluster' and 'cluster'. When we speak of 'ONOS cluster', we specifically mean all ONOS instances in the cluster, excluding anything else. In 'cluster' we mean the whole cluster, for example with all ONOS instances, all Atomix

instances and consensus status[6].
Another important wording is 'immediate', which means the described transition is modelled with an instantaneous transition, with reference to its description in Chapter 2.

ONOS elements

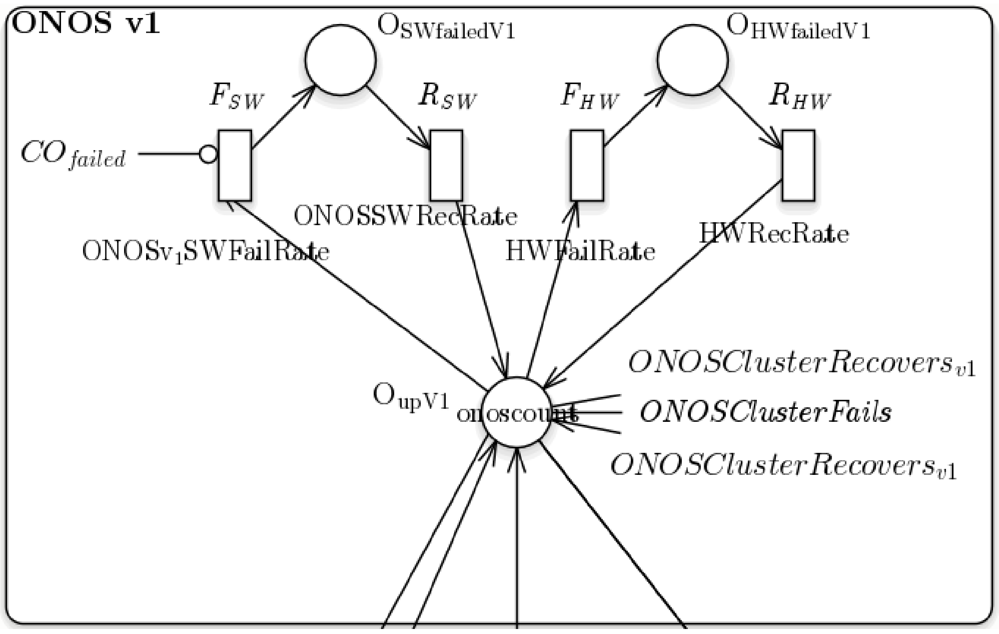


Figure 4.2: The white ONOS square in GreatSPN

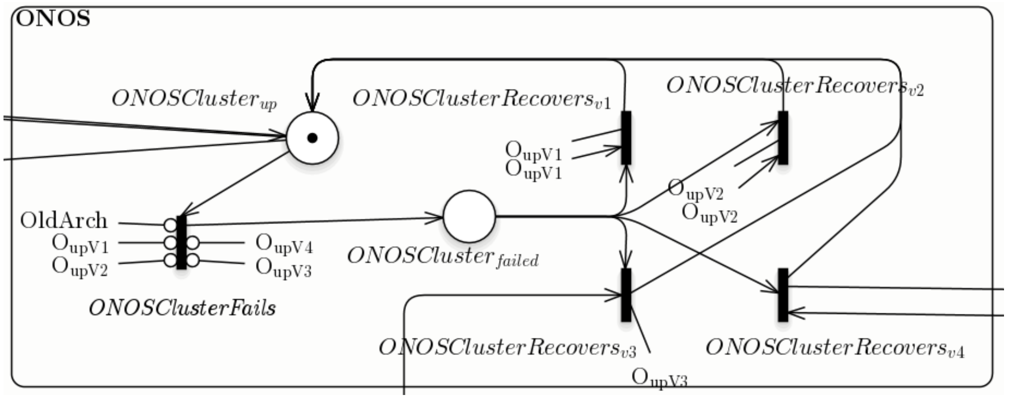


Figure 4.3: The white ONOS cluster square in GreatSPN

[6]Note that the model per default presents the new architecture in which the difference between 'ONOS cluster' and 'cluster' exists.

We have multiple **white squares**, one per ONOS version and one to present the current availability of the ONOS cluster.
In each ONOS version square, like the ONOS v1 square in Figure 4.2, we can find the following:

versioned O_{up} Place : Holds tokens that represent *up* ONOS instances.

versioned $O_{SWfailed}$ Place : Holds tokens that represent *failed* ONOS instances due to software failure.

versioned $O_{HWfailed}$ Place : Holds tokens that represent *failed* ONOS instances due to hardware failure.

F_{SW} Transition : This transition represents a software failure of an ONOS instance. When it fires, it moves one token from the versioned O_{up} place into the versioned $O_{SWfailed}$ place. Additionally, it has an inhibitor arc to CO_{failed} and its rate is defined by the versioned $ONOSSWFailRate$ parameter.

R_{SW} Transition : This transition represents a software recovery of an ONOS instance. When it fires, it moves one token from the versioned $O_{SWfailed}$ place into the versioned O_{up} place. Its rate is defined by the $ONOSSWRecRate$ parameter.

F_{HW} Transition : This transition represents a hardware failure of an ONOS instance. When it fires, it moves one token from the versioned O_{up} place into the versioned $O_{HWfailed}$ place. Its rate is defined by the $HWFailRate$ parameter.

R_{HW} Transition : This transition represents a hardware recovery of an ONOS instance. When it fires, it moves one token from the versioned $O_{HWfailed}$ place into the versioned O_{up} place. Its rate is defined by the $HWRecRate$ parameter.

In the ONOS cluster status square in Figure 4.3 we can find:

$ONOSCluster_{up}$ Place : Holds at most one token, representing that at least one ONOS instance is *up*.

$ONOSCluster_{failed}$ Place : Holds at most one token, representing that no ONOS instance is *up*.

$ONOSClusterFails$ Transition : This transition represents the move to unavailability for the ONOS cluster. When it fires, it moves the token from $ONOSCluster_{up}$ to

$ONOSCluster_{failed}$. Additionally, it has inhibitor arcs to $OldArch$ and all versions of O_{up}. Its rate is immediate.

versioned $ONOSClusterRecovers$ **Transition** : This transition represents the move to availability for the ONOS cluster. When it fires, it moves the token from $ONOSCluster_{failed}$ to $ONOSCluster_{up}$. Additionally, it has as one input and output arc to the respective versioned O_{up} place with the multiplicity of 1. Its rate is immediate.

Atomix elements

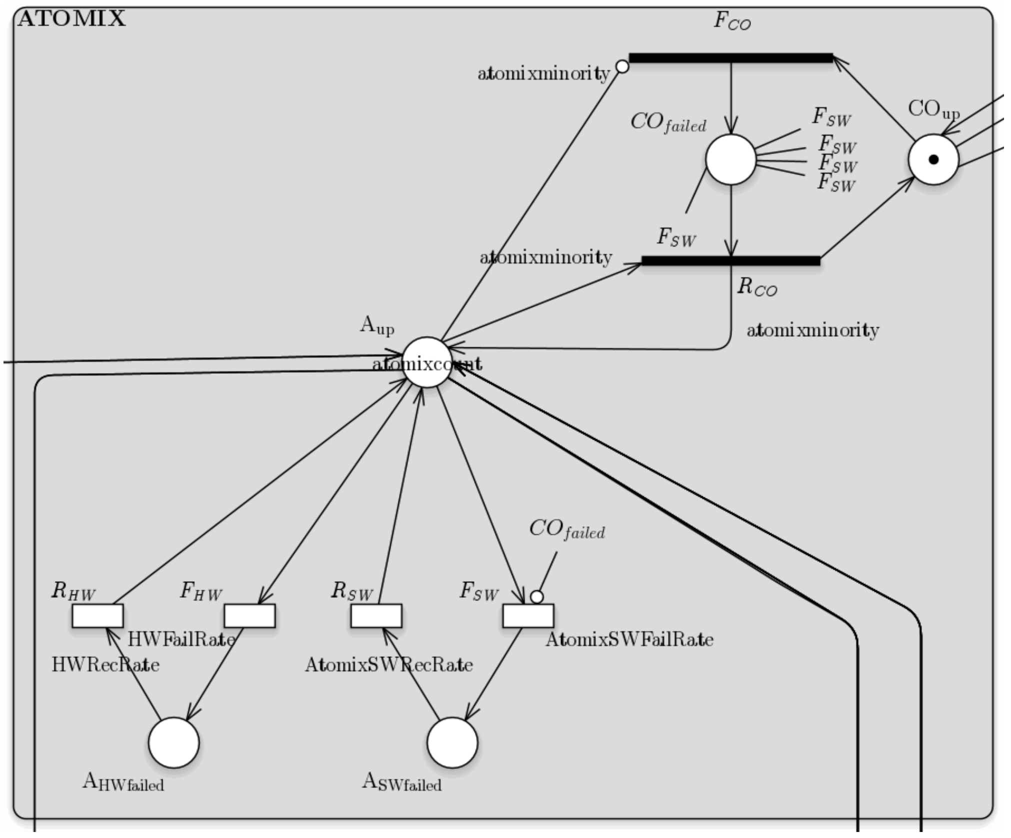


Figure 4.4: The grey Atomix square in GreatSPN

The **grey square** in Figure 4.4 contains Atomix elements:

A_{up} **Place** : Holds tokens that represent *up* Atomix instances.

$A_{SWfailed}$ **Place** : Holds tokens that represent *failed* Atomix instances due to software failure.

$A_{HWfailed}$ **Place** : Holds tokens that represent *failed* Atomix instances due to hardware failure.

CO_{up} **Place** : Holds at most one token, representing that consensus is *up*.

CO_{failed} **Place** : Holds at most one token, representing that consensus is *failed*.

F_{SW} **Transition** : This transition represents a software failure of an Atomix instance. When it fires, it moves one token from the A_{up} place into the $A_{SWfailed}$ place. Additionally, it has an inhibitor arc to CO_{failed} and its rate is defined by the $AtomixSWFailRate$ parameter.

R_{SW} **Transition** : This transition represents a software recovery of an Atomix instance. When it fires, it moves one token from the $A_{SWfailed}$ place into the A_{up} place. Its rate is defined by the $AtomixSWRecRate$ parameter.

F_{HW} **Transition** : This transition represents a hardware failure of an Atomix instance. When it fires, it moves one token from the A_{up} place into the $A_{HWfailed}$ place. Its rate is defined by the $HWFailRate$ parameter.

R_{HW} **Transition** : This transition represents a hardware recovery of an Atomix instance. When it fires, it moves one token from the $A_{HWfailed}$ place into the A_{up} place. Its rate is defined by the $HWRecRate$ parameter.

F_{CO} **Transition** : This transition represents the failure of the consensus protocol. When it fires, it moves one token from the CO_{up} place into the CO_{failed} place. Additionally, it has an inhibitor arc to A_{up} with the multiplicity of $atomixminority$, whereas $atomixminority$ is equal to the rounded down result of $\frac{atomixcount}{2+1}$. Its rate is immediate.

R_{CO} **Transition** : This transition represents the recovery of the consensus protocol. When it fires, it moves one token from the CO_{failed} place into the CO_{up} place. Additionally, it has one input and output arc to A_{up} with the multiplicity of $atomixminority$ each. Its rate is immediate.

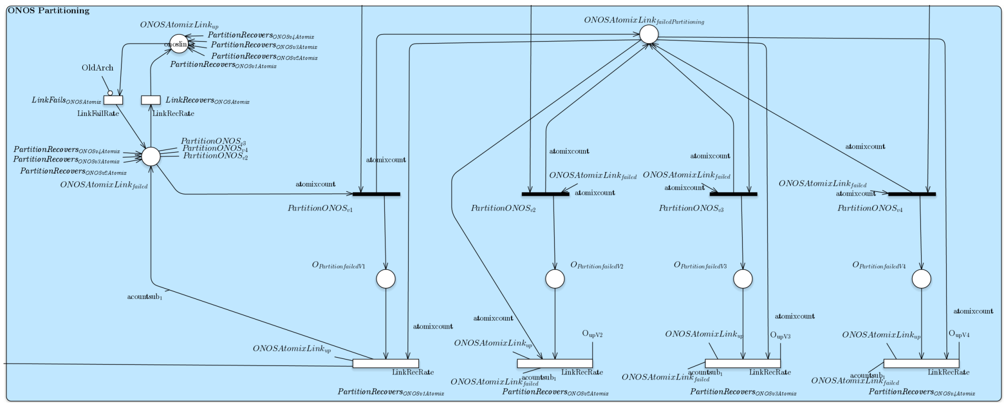


Figure 4.5: The blue Partition square for ONOS instances in GreatSPN

Network Partition elements

Next up are the **blue squares**, they contain the network partition elements.
One square, Figure 4.6, is for Atomix:

$AtomixLink_{up}$ **Place** : Holds tokens that represent *up* Atomix links.

$AtomixLink_{failed}$ **Place** : Holds tokens that represent *failed* Atomix links.

$A_{Partitionfailed}$ **Place** : This place holds tokens that represent *failed* Atomix instances due to network partitioning.

$AtomixLink_{failedPartitioning}$ **Place** : This place exits two times, once in combination with $PartitionLastAtomixInstances$ and once in combination with $PartitionAtomixInstance$. In both variants it holds tokens that represent *failed* Atomix links that are also part of a partition.

$LastAtomixPartitioned$ **Place** : This place holds zero or two tokens, representing the last two *failed* Atomix instances due to network partitioning.

$LinkFail_{Atomix}$ **Transition** : This transition represents the failure of an Atomix link. When it fires, it moves a token from $AtomixLink_{up}$ to $AtomixLink_{failed}$. Its rate is defined by the $LinkFailRate$ parameter.

$LinkRecovers_{Atomix}$ **Transition** : This transition represents the recovery of an Atomix link. When it fires, it moves a token from $AtomixLink_{failed}$ to $AtomixLink_{up}$. Its rate is defined by the $LinkRecRate$ parameter.

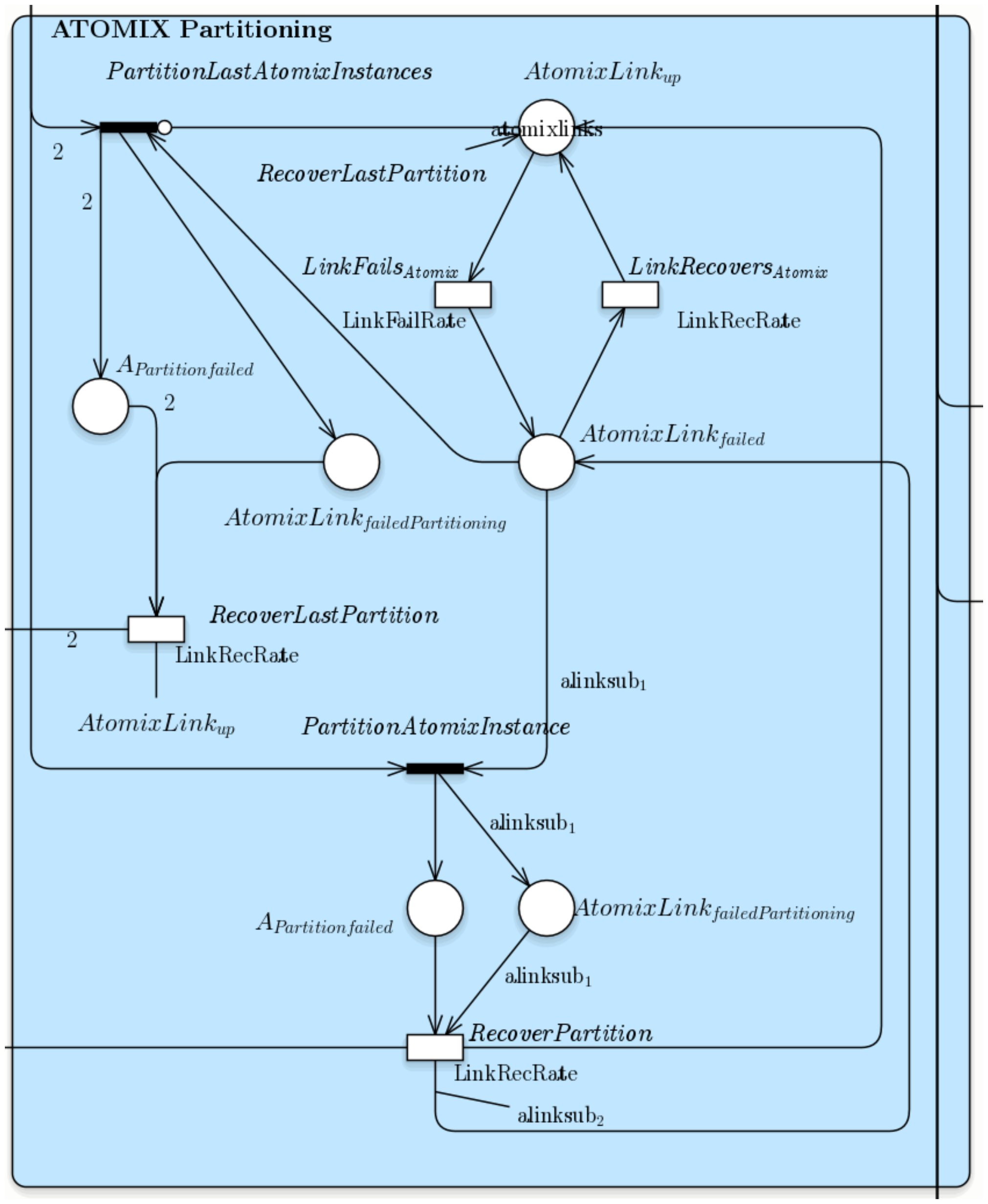


Figure 4.6: The blue Partition square for Atomix instances in GreatSPN

$PartitionAtomixInstance$ **Transition** : This transition represents a partition failure of an Atomix instance. When it fires, it moves $atomixcount - 1$ ($alinksub1$) tokens from $AtomixLink_{failed}$ to $AtomixLink_{failedPartitioning}$ and one token from A_{up} to $A_{Partitionfailed}$. Its rate is immediate.

$RecoverPartition$ **Transition** : This transition represents a partition recovery of an Atomix instance. When it fires, it moves $atomixcount - 2$ ($alinksub2$) tokens from $AtomixLink_{failedPartitioning}$ to $AtomixLink_{failed}$, one token into $AtomixLink_{up}$ and one token from $A_{Partitionfailed}$ to A_{up}. Its rate is defined by the $LinkRecRate$ parameter.

$PartitionLastAtomixInstances$ **Transition** : This transition represents the partition failure of the last two *up* Atomix instances. When it fires, it moves a token from $AtomixLink_{failed}$ to $AtomixLink_{failedPartitioning}$ and two tokens from A_{up} to $LastAtomixPartitioned$. Additionally, it has an inhibitor arc to $AtomixLink_{up}$ and its rate is immediate.

$RecoverLastPartition$ **Transition** : This transition represents the partition recovery of the two Atomix instances mentioned above. When it fires, it moves the token from $AtomixLink_{failedPartitioning}$ to $AtomixLink_{up}$ and the two tokens from $LastAtomixPartitioned$ to A_{up}. Its rate is defined by the $LinkRecRate$ parameter.

As a separate paragraph we now explain the reason why the partitioning of the last two Atomix instances is considered differently than the partitioning of all Atomix instances before.
Let's assume a network like presented in Figure 4.1. If two Atomix links fail, the first Atomix instance is partitioned. Now we cannot continue like before, waiting for two Atomix links to fail, as there is only one Atomix link left. So, if the last Atomix link fails, the last two Atomix instances are partitioned. In the model this is done via the transition $PartitionLastAtomixInstances$. The recovery of these two instances is considered with the $RecoverLastPartition$ transition.

And the blue square on the right side of the model, Figure 4.5, is for ONOS:

$ONOSAtomixLink_{up}$ **Place** : Holds tokens that represent *up* ONOS-Atomix links.

$ONOSAtomixLink_{failed}$ **Place** : Holds tokens that represent *failed* ONOS-Atomix links.

$ONOSAtomixLink_{failedPartitioning}$ **Place** : Holds tokens that represent *failed* ONOS-Atomix links that are also part of a partition.

versioned $O_{Partitionfailed}$ **Place** : Holds tokens that represent *failed* ONOS instances due to network partition.

$LinkFails_{ONOSAtomix}$ **Transition** : This transition represents the failure of an ONOS-Atomix link. When it fires, it moves a token from $ONOSAtomixLink_{up}$ to $ONOSAtomixLink_{failed}$. Additionally, it has an inhibitor arc to *OldArch* and its rate is defined by the *LinkFailRate* parameter.

$LinkRecovers_{ONOSAtomix}$ **Transition** : This transition represents the recovery of an ONOS-Atomix link. When it fires, it moves a token from $ONOSAtomixLink_{failed}$ to $ONOSAtomixLink_{up}$. Its rate is defined by the *LinkRecRate* parameter.

versioned *PartitionONOS* **Transition** : This transition represents the partition failure of an ONOS instance. When it fires, it moves *atomixcount* tokens from $ONOSAtomixLink_{failed}$ to $ONOSAtomixLink_{failedPartitioning}$ and one token from the versioned O_{up} to the versioned $O_{Partitionfailed}$ place. Its rate is immediate.

versioned $PartitionRecovers_{ONOSAtomix}$ **Transition** : This transition represents the partition recovery of an ONOS instance. When it fires, it moves $atomixcount - 1$ tokens from $ONOSAtomixLink_{failedPartitioning}$ to $ONOSAtomixLink_{failed}$, one token into $ONOSAtomixLink_{up}$ and one token from the versioned $O_{Partitionfailed}$ to the versioned O_{up} place. Its rate is defined by the *LinkRecRate* parameter.

Upgrade elements

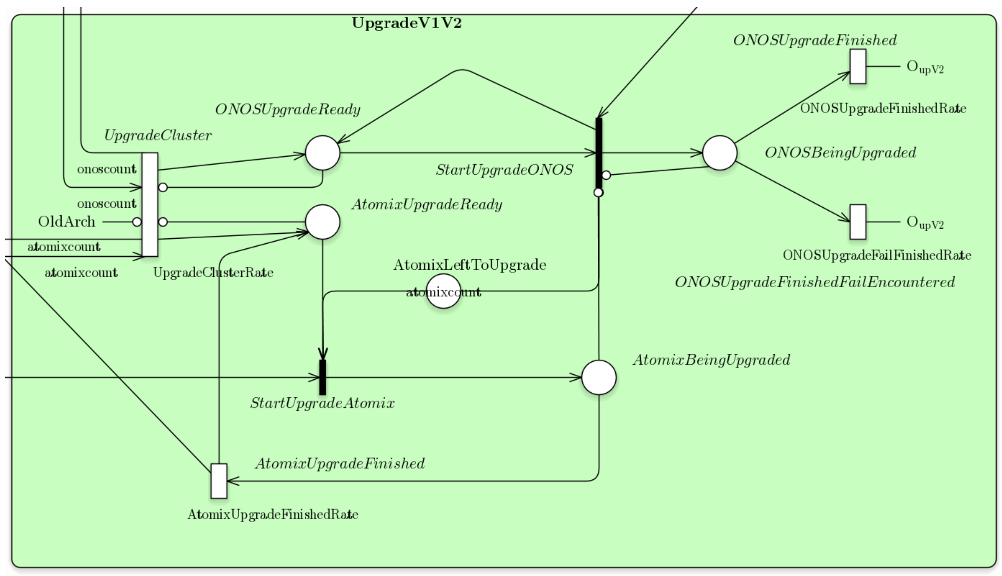


Figure 4.7: The green upgrade (v1 to v2) square in GreatSPN

The **green squares** contain the upgrade process, one for each version upgrade. The square for the upgrade from version 1 to version 2 can be seen in Figure 4.7. All green squares have the same places and transitions. The multiple green upgrade squares only differ in the connected versioned O_{up} places.

$ONOSUpgradeReady$ **Place** : Holds at most one token, representing that the next ONOS instance can be upgraded.

$AtomixUpgradeReady$ **Place** : Holds at most one token, representing that the next Atomix instance can be upgraded.

$ONOSBeingUpgraded$ **Place** : Holds at most one token, representing the currently upgraded ONOS instance.

$AtomixLeftToUpgrade$ **Place** : Holds at most *atomixcount* tokens, representing the number of Atomix instances that are left to upgrade.

$AtomixBeingUPgraded$ **Place** : Holds at most one token, representing the currently upgraded Atomix instance.

UpgradeCluster **Transition** : This transition represents the start of a cluster upgrade. When it fires, it moves a token into *ONOSUpgradeReady* and *AtomixUpgradeReady*. Additionally, it has one input and output arc to the respective O_{up} place with the multiplicity of *onoscount*. Analogue to that is the connection to A_{up} with the multiplicity of *atomixcount*. It also has three inhibitor arcs to *OldArch*, *ONOSUpgradeReady* and *AtomixUpgradeReady* and its rate is defined by the *UpgradeClusterRate* parameter.

StartUpgradeONOS **Transition** : This transition represents a starting ONOS upgrade. When it fires, it removes the token from *ONOSUpgradeReady*, and it moves one token from the versioned O_{up} place to *ONOSBeingUpgraded*. Its rate is immediate.

ONOSUpgradeFinished **Transition** : This transition represents a finished ONOS upgrade. When it fires, it moves the token from *ONOSBeingUpgraded* to the versioned O_{up} place and inserts a token into *ONOSUpgradeReady*. Its rate is defined by the *ONOSUpgradeFinishedRate*.

ONOSUpgradeFinishedFailEncountered **Transition** : This transition represents a finished ONOS upgrade with encountered and handled failures. When it fires, it moves the token from *ONOSBeingUpgraded* to the versioned O_{up} and inserts a token into *ONOSUpgradeReady*. Its rate is defined by the *ONOSUpgradeFinishedFailEncounteredRate*

StartUpgradeAtomix **Transition** : This transition represents a starting Atomix upgrade. When it fires, it removes one token from *AtomixUpgradeReady* & *AtomixLeftToUpgrade*, and it moves one token from A_{up} to *AtomixBeingUpgraded*.

AtomixUpgradeFinished **Transition** : This transition represents a finished Atomix upgrade. When it fires, it moves the token from *AtomixBeingUpgraded* to A_{up} and inserts a token into *AtomixUpgradeReady*.

Cluster Metadata elements

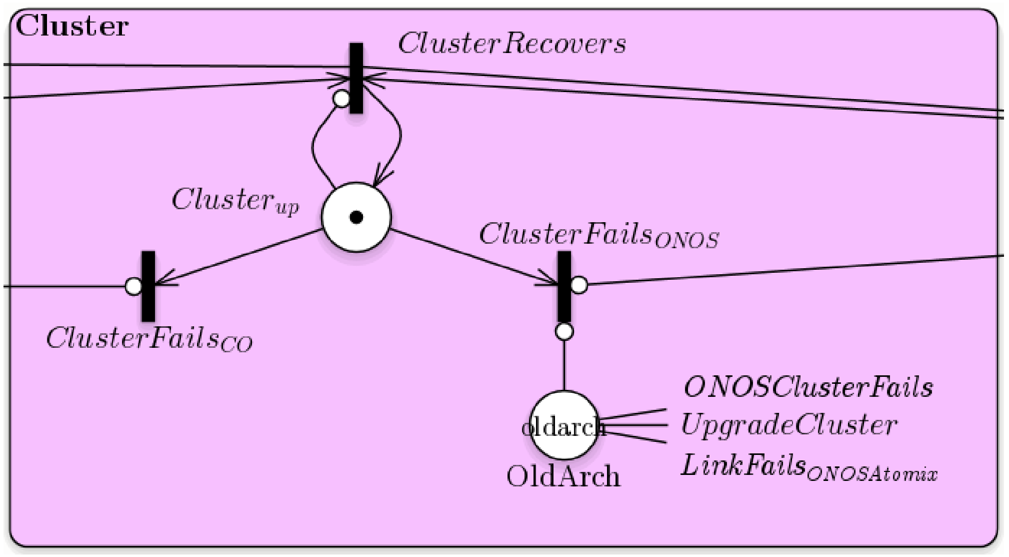


Figure 4.8: The pink cluster metadata square in GreatSPN

In Figure 4.8 we can find the **pink square** with the cluster metadata:

$Cluster_{up}$ **Place** : Holds at most one token, representing that the cluster is available.

OldArch **Place** : If at least one token is set, it signalizes that the current model and its set parameters represent the old architecture. The token has to be manually set, and it makes no difference if more than one token is set.

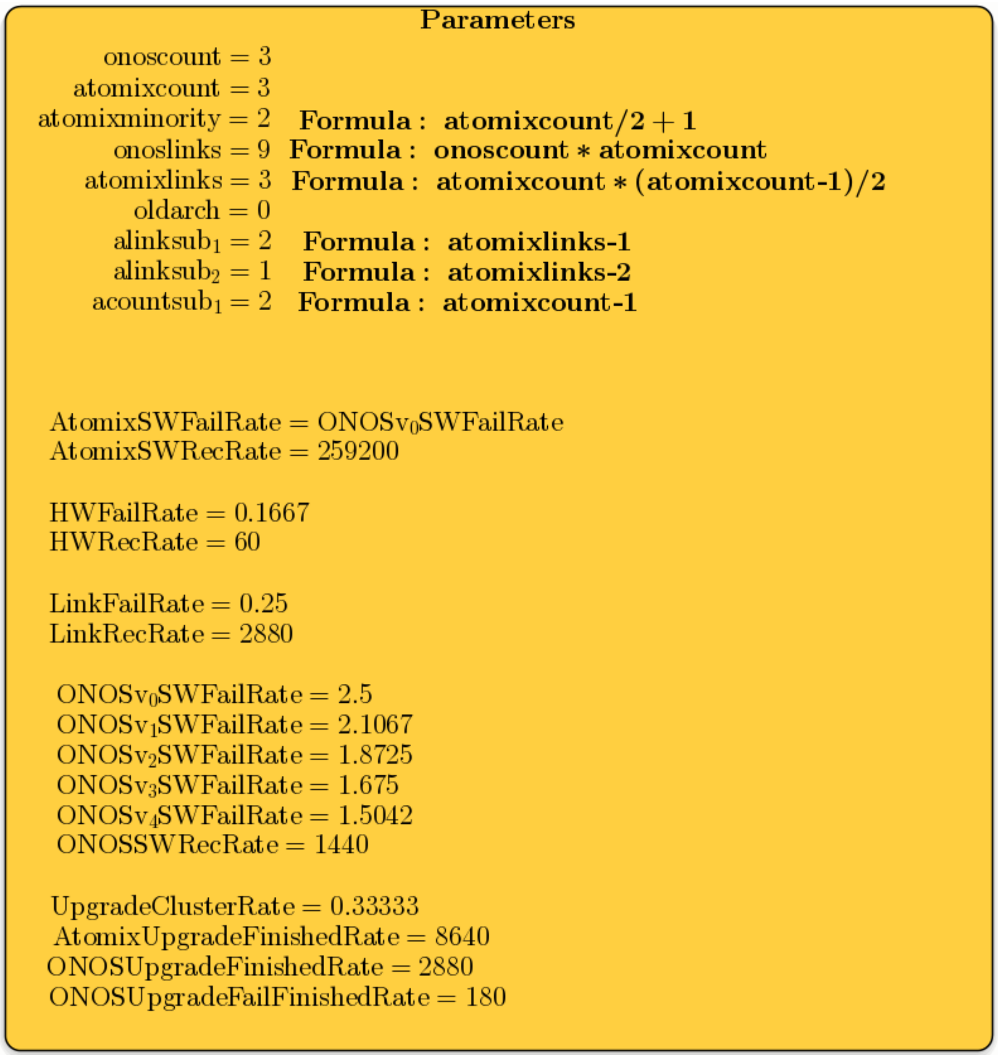
$ClusterFails_{ONOS}$ **Transition** : This transition represents the move to unavailability for the cluster due to a *failed* ONOS cluster. When it fires, it removes the token from $Cluster_{up}$. Additionally, it has inhibitor arcs to *OldArch* and $ONOSCluster_{up}$ and its rate is immediate.

$ClusterFails_{CO}$ **Transition** : This transition represents the move to unavailability for the cluster due to *failed* consensus. When it fires, it removes the token from $Cluster_{up}$. Additionally, it has an inhibitor arc to CO_{up} and its rate is immediate.

ClusterRecovers **Transition** : This transition represents the move to availability for the cluster due to an *up* ONOS cluster and *up* consensus. When it fires, it inserts a token into $Cluster_{up}$. Additionally, it has input and output arcs to both

$ONOSCluster_{up}$ and CO_{up}, and it has an inhibitor arc to $Cluster_{up}$. Its rate is immediate.

Model Parameters



Parameters

onoscount = 3	
atomixcount = 3	
atomixminority = 2	**Formula : atomixcount/2 + 1**
onoslinks = 9	**Formula : onoscount ∗ atomixcount**
atomixlinks = 3	**Formula : atomixcount ∗ (atomixcount-1)/2**
oldarch = 0	
$\text{alinksub}_1 = 2$	**Formula : atomixlinks-1**
$\text{alinksub}_2 = 1$	**Formula : atomixlinks-2**
$\text{acountsub}_1 = 2$	**Formula : atomixcount-1**

$\text{AtomixSWFailRate} = \text{ONOSv}_0\text{SWFailRate}$
AtomixSWRecRate = 259200

HWFailRate = 0.1667
HWRecRate = 60

LinkFailRate = 0.25
LinkRecRate = 2880

$\text{ONOSv}_0\text{SWFailRate} = 2.5$
$\text{ONOSv}_1\text{SWFailRate} = 2.1067$
$\text{ONOSv}_2\text{SWFailRate} = 1.8725$
$\text{ONOSv}_3\text{SWFailRate} = 1.675$
$\text{ONOSv}_4\text{SWFailRate} = 1.5042$
ONOSSWRecRate = 1440

UpgradeClusterRate = 0.33333
AtomixUpgradeFinishedRate = 8640
ONOSUpgradeFinishedRate = 2880
ONOSUpgradeFailFinishedRate = 180

Figure 4.9: The used parameters and their default values in GreatSPN

At last, we present in Figure 4.9 the used parameters and their default values according to Section 4.3. Even though GreatSPN can handle formulas as a link multiplicity, e.g. $atomixcount - 1$, the upcoming tool used for automated simulation cannot. So we needed to fixate all parameter values, including the ones we could calculate, which for example explains the existence of the $alinksub1$ parameter that otherwise could be replaced with the simple $atomixlink - 1$ formula where needed.

One model for both architectures

The possibility of the two architectures being modeled in one GSPN has the benefit that changes are not needed to be applied in two different files, which could lead to inconsistencies, e.g. if the copying from one to the other model file is forgotten.

Due to the separated ONOS and Atomix instances, the shown GSPN model represents the new architecture per default, but we can also represent the old ONOS architecture with it by taking the following steps. The first step is to set one token into the *OldArch* place. The next one is to set the $ONOScount$ parameter to zero and the *AtomixCount* parameter to the count of wanted ONOS instances. At last, we need to adjust the failure and recovery rates of Atomix by setting them to the rates for the old architecture ONOS instances. In our case, in regard to the used parameters in Section 4.3, we only need to adjust the *AtomixSWRecRate* parameter to the value of $ONOSSWRecRate$. The *AtomixSWFailRate* is already set to the failure rate of ONOS v1.13, which is realized with the $ONOSv0SWFailRate$ parameter.

4.4.2 Move to SWNs

We could have refactored our model with the use of SWNs. For example by merging multiple places (e.g. the versioned O_{up}) into one, because the identity a place gives an anonymous token is then replaced by the coloured token itself. Since this change would maybe only increase the visual clarity of the model, which we think is good enough right now, we are not switching to SWNs.

4.5 Chapter Conclusion

In this chapter we formulated and reasoned our research question in Section 4.1, we reasoned the considered elements of our model in Section 4.2 and the used parameter values in Section 4.3. In the last Section 4.4 we presented the GSPN model and its details.

The next Chapter 5 builds heavily upon this chapter, in it, we will validate our realized model and its results, so we can evaluate our model in Chapter 6.

5 Validation

In this chapter we will validate our model in three major ways. First we will validate the realization of our model from Section 4.4 in Section 5.1. Then we will describe manual simulations in Section 5.2 and automatic simulations in Section 5.3. The latter is used for our sensitivity analysis in Section 5.4 and plausibility analysis in Section 5.5.

5.1 Model Realization

The goal of this section is to provide arguments, that the specified behaviour in Sections 4.2.1 and 4.2.2 is correctly realized in the model presented in Section 4.4. This section extends Section 4.4.1 and the explained terms are reused here.

In the following itemisation we will reason how and if the included elements from Section 4.2.1 are realized.

EO1, ONOS instances : *up* ONOS instances and their respective version are realized with tokens in the versioned O_{up} places.

EO2, Atomix instances : *up* Atomix instances are considered with tokens in the A_{up} place.

EO3, Control plane links : We modelled the *up* control plane links with tokens in the $ONOSAtomixLink_{up}$, $AtomixLink_{up}$ and *failed* links in their associated *failed* places.

EO4, Consensus status : The *up* consensus status is included in the model with the CO_{up} place, the *failed* consensus status is considered with the CO_{failed} place.

EO5, Cluster availability : Cluster availability is incorporated into the model with the $Cluster_{up}$ place, if it contains a token the cluster is available, otherwise it is not.

EB1, ONOS and Atomix failures : ONOS and Atomix failures are represented by the respective failures due to hardware (F_{HW} transition, $A_{HWfailed}$ place and versioned $O_{HWfailed}$ places) and software (F_{SW} transition, $A_{SWfailed}$ place and versioned $O_{SWfailed}$ places). One node per instance is considered, as each Atomix or ONOS instance token can fail due to hardware, independent of the other instances' status.

The modelling of the network partitioning includes the $PartitionAtomixInstance$, $PartitionLastAtomixInstances$ transitions & $A_{Partitionfailed}$, $PartitionLastAtomixInstances$ places for Atomix and the versioned $PartitionONOS$ transitions & versioned $O_{Partitionfailed}$ places for ONOS.

All failures are recoverable via their respective recovery transition, software (R_{SW}), hardware (R_{HW}) and network partition ($RecoverPartition$, $RecoverLastPartition$ and versioned $PartitionRecovers_{ONOSAtomix}$).

EB2, ONOS and Atomix idle : That all instances are idling while the consensus is *failed* is considered by having an inhibitor arc from CO_{failed} to all F_{SW} transitions which prevents these transitions to fire in this failure case.

EB3, Combined failure of ONOS and Atomix in old architecture : In the old architecture we only fill the A_{up} place with token that represent coupled Atomix and ONOS instances, as described in Section 4.4.1. As there is one token for both Atomix and ONOS, they only can fail and recover together.

EB5, Upgrade of cluster in new architecture : The upgrade of the cluster in the new architecture is started with the *UpgradeCluster* transition for each new version. Each upgrade takes tokens from the current ONOS version, e.g. O_{upV1}, and eventually moves them into the next ONOS version, e.g. O_{upV2}. The effect of the upgrade onto ONOS' availability is dependent on the used parameters for the different ONOS versions, so according to the current simulation, the upgrade can improve, worsen or not impact ONOS' availability for each ONOS version independently.

In the following itemisation we explain how the excluded elements from Section 4.2.2 are not considered in the model.

EO6, No data plane : The realization does not consider any SDN switches, hosts or date plane links. Nor do we consider any data plane failures and recoveries in our cluster availability.

EB6, 100% failure detection : If a failure occurs it has a direct impact, e.g. with F_{SW} that moves a token from A_{up} to $A_{SWfailed}$.

EB7, No load related failures : Load is not considered in the failure rates, as it can be seen in Section 4.3, nor do we model special failure transitions that are specific for load failures.

EB8, Failure stacking : Once an instance is *failed*, it cannot fail anymore. Only once it has recovered the next failure can happen. For example a token from $A_{SWfailed}$ (Atomix instance failed due to software) has no transition to directly move into $A_{HWfailed}$ (failed due to hardware), only after it recovers from its software failure via R_{SW} and the token is in A_{up}, it can fail due to F_{HW} and move into $A_{HWfailed}$.

EB9, Atomix upgrade does not improve its availability : This easily be seen in the difference of the ONOS squares (white) and the Atomix square (grey). The Atomix square does not contain versioned A_{up} places, neither any versioned failure transitions with different failure rates. Atomix instances always fail / recover with the same rate as they always use the same transitions and parameters.

EB10, No upgrade in old architecture : As explained in Section 4.4.1, we place a token in *OldArch* when we want to simulate the old architecture. The token in this place and the inhibitor arc from it to the first *UpgradeCluster* transition, prevents the first upgrade. All other upgrades are prevented as an upgrade needs *onoscount* tokens in the versioned O_{up} places, but the O_{upV2} and O_{upV3} places will always be empty.

EB11, No gossip protocol : The parameters described in Section 4.3 do not consider the gossip protocol, neither does it any place nor transition as presented in Section 4.4.1.

EB12, No security related unavailability : Analogue to **EB11**, no parameter, place or transition does consider security related failures and recoveries.

5.2 Manual Simulation

Especially during development of the model, we started manual simulations within GreatSPN to verify if we modelled correctly. This way we can test if the model behaves as

expected in certain situations. An example use case is to check if tokens are duplicating or diminishing, e.g. when a $UpgradeCluster$ transition fires, we take $atomixcount$ tokens from A_{up} and directly put them back. With manual simulations we can force this situation, manually fire the transition and check if A_{up} still has $atomixcount$ tokens afterwards.
This way we could identify and locate wrong realizations in the model, essentially debugging it. Once we felt confident that no error is left, we began with automated simulations.

5.3 Automated Simulations

Additional to the manual approach, GreatSPN features tools to analyze performance aspects of a GSPN or SWN model, e.g. with the 'WNSIM' tool mentioned in Chapter 3 which was used by [55, 22].
The results of a WNSIM simulation are the mean number of tokens for each place and the mean throughput for each transition throughout the simulation time. As our cluster steady state availability, described in Section 2.1, we take the mean number of tokens in the $Cluster_{up}$ place. This is possible since the number of tokens is at most one, as described in Sections 4.2.1 and 4.4, which can represent a percentage with its value between 0 and 1. To get this value, we do not have to wait for the simulation to finish as it constantly creates fine-grained logs from which one can gather the cluster availability. This allows us to interrupt simulations if they are running too long.
For all simulations we use a confidence of 90% and an approximation of 50%. Lower approximation leads to more precise results, but also to immensely increased simulation times. With the current value we have simulations that do not finish within six hours. This time is not negligible, as we execute for example 160 simulations[1] for the upcoming $HWFailRate$ parameter sensitivity analysis alone.

The simulations are executed on the same hardware that was used in Section 4.3. We can call the 'WNSIM' tool from the command line via `/usr/local/GreatSPN/bin/WNSIM` and via optional arguments we can overwrite the default parameter values in the model, as explained in Section 4.3 and shown in Section 4.4.1. Our fully automated simulation and results gathering tool works like:

1. Read the configuration files, these can include new parameter values

[1] $architectures * stepcount * repititions = 2 * 16 * 5$

2. Start the simulations
    a) Sequentially or parallel with an optional limit of concurrent simulations
    b) Generate a new seed for each simulation
    c) Timeout simulation after an optional given time
3. Waiting for all simulations to finish
4. Gather the fine-grained results from all simulations
5. Create one *xlsx* file with one worksheet per parameter and create charts

For the interested reader, we added more description of our simulation tool in the appendix Section A.2.

5.4 Sensitivity Analysis

For our sensitivity analysis, we interrupt a simulation after one hour. Each sensitivity analysis analyzes one parameter. All others parameters are unaltered, see Sections 4.3 and 4.4.1 for parameter default values. The value of an analyzed parameter is altered with each 'step'. We configured our steps to be between -5 and 10 inclusive, each step differs $10\% * step$ from the default value which is used in step 0. For example with the $ONOSSWRecRate$ parameter we have the different values per step:

Step -5, -50% : $1440 * 0,5 = 720$

Step -4, -40% : $1440 * 0,6 = 864$

...

Step -1, -10% : $1440 * 0,9 = 1296$

Step 0, +0% : $1440 * 1 = 1440$ (default value)

Step 1, +10% : $1440 * 1,1 = 1584$

Step 2, +20% : $1440 * 1,2 = 1728$

...

Step 9, +90% : $1440 * 1,9 = 2736$

Step 10, +100% : $1440 * 2 = 2880$

For parameters with integer values, the difference is $1 * stepno$. Each step is repeated five times. For each repetition we generate a new seed which is the value of BASHs `$RANDOM` variable at that moment.
So from a different perspective, each analyzed parameter, step, repetition and architecture is a separate simulation with a unique seed. As an optimization, not all parameters are executed in all architecture. For example, the upgrade parameters are only analyzed in the new architecture as they have no impact in the old architecture.

As a side note, we originally planned to only simulate and analyze the sensitivity of selected parameters which in turn means that we could have missed interesting points. This decision was revised once we thought of automating this process. Due to the high degree of automation we can easily analyze all parameters to see if some of them have an interesting course throughout their steps.

In all simulations, the step difference did not propagate to the cluster availability. In other words, when the value of a parameter was increased by 10%, the cluster availability did change by less than 10%. We present one sensitivity result in more depth, as an example how these results have to be read and interpreted.
In Figure 5.5 we can see the mean result of multiple sensitivity analyzes of the *UpgradeClusterRate* parameter. For the sensitivity analysis it is not important what this parameter represents, just how much it influences the overall result when its value changes. The results have a blue line if the sensitivity analysis was done only once, and it is green like in our example if the analysis was repeated multiple times, the green line then represents the mean.
Like explained above, the sensitivity analysis has 16 steps, from -5 to 10 inclusive, three steps and their meaning are described next:

Step 0 : The base value of the sensitivity analysis is always step 0 as it represents results with the default values of our model from Section 4.3. In this case step 0 results in a cluster availability of $0,9999872$ ($99,99872\%$).

Step -5 : The parameter has -50% of its default value ($1440 * 0,5$). This results in a cluster availability of $0,9999817$. This is a difference to our base cluster availability of about $0,00055\%$, this is far from -50%.

Step 4 : The parameter has $+40\%$ of its default value $(1440 * 1,4)$. This results in a cluster availability of $0,9999904$. This is a difference to our base cluster availability of about $0,00032\%$, which is far away from $+40\%$.

So the change of the parameter value did influence to the cluster availability, but not by the same percentage.

This can be observed for all other results too. More result examples like the one just described. Some of them will be discussed multiple times in the upcoming sections and chapters, the figures are placed next to the discussion that depends the most on it.

5.5 Simulation Results Plausibility

Besides looking at sensitivity we are now looking at the plausibility of some simulation results, partly already created during sensitivity analysis.

In contrast to the sensitivity analysis, it is very important for the plausibility analysis to consider the context of the analyzed parameter, as different parameters have different impact on the cluster's availability. For example an increased recovery rate is expected to increase the cluster availability while an increased failure rate is expected to decrease it.

5.5.1 Basic Architecture Results

For the basic architecture results, we take both unaltered architectures, execute each five times and calculate the mean of all repetitions. Like a sensitivity analysis without steps. The simulation results are an availability of 99,9981% for the new and 99,99550% for the old architecture.
These results are just slightly above of the 99,99% availability that ONOS specified themselves [5, 13]. Keep in mind that these sources are based upon an older version of ONOS in the old architecture. With reference to Table 4.1, ONOS' availability most certainly improved over the last few years, so we are confident that the basic architecture results are plausible.

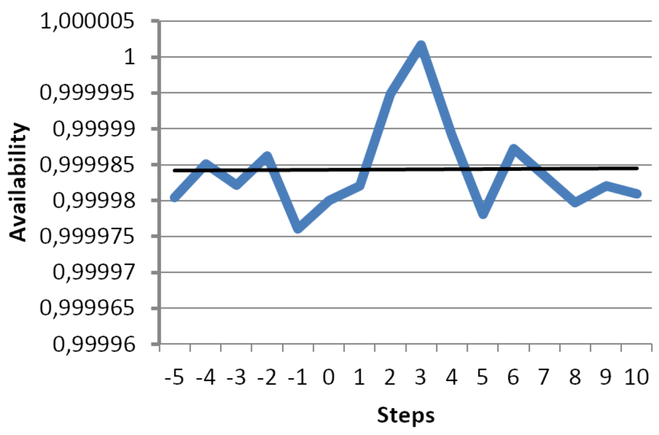


Figure 5.1: Cluster availability (blue) and linear trend line (black) of the $ONOSv2SWFailRate$ parameter steps in the new architecture

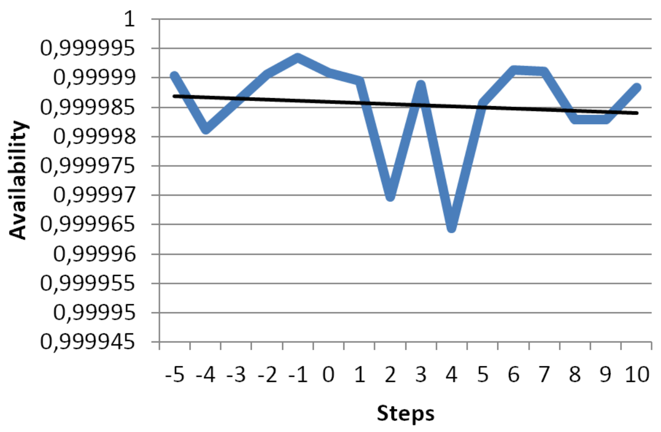


Figure 5.2: Cluster availability (blue) and linear trend line (black) of the $ONOSv3SWFailRate$ parameter steps in the new architecture

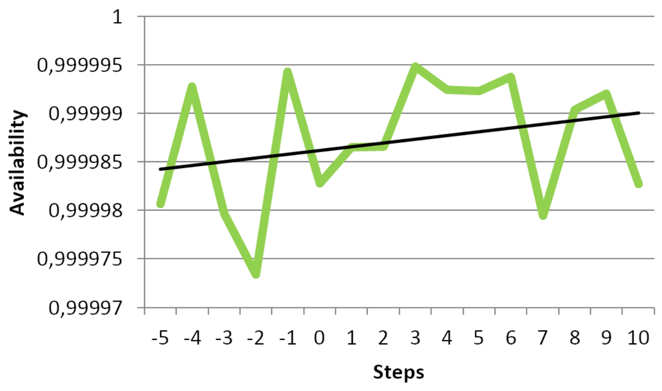


Figure 5.3: Cluster availability (green) and linear trend line (black) of the *AtomixSWFail* parameter steps in the new architecture

5.5.2 Failure Rates

The **Atomix software failure rate** results in Figure 5.3 do only consider the new architecture, as in the old architecture this parameter has the value of the ONOS v1.13 software failure rate as described in Section 4.4.1. We executed this sensitivity analysis multiple times, since the cluster availability improves when the Atomix software failure rate increases and this is counterintuitive. When an instance is more likely to fail, we expect it to generate more unavailability for the whole cluster. We wanted to make sure that this result is not based upon coincidence, e.g. a bad seed. The retries confirmed the first result, this can be said about all sensitivity analysis repetitions (green lines).
We assume, that the cluster improves as the Atomix instance is more likely to fail due to software than to hardware or partitioning and this improves the cluster availability, as a software failure is faster recovered than the other two.

In the Figure 6.3 we can observe the **hardware failure rate** results. The results match our expectations, as the increasing failure rate decreases the cluster availability in both architectures. The same is true for the **software failure rate of ONOS v2 and v3** and **link failure** results in Figures 5.1, 5.2 and 6.1.

5.5.3 Recovery Rates

In Figure 5.4 we can observe that sometimes the results are very inconsistent as the results line takes rapid changes. This could be a hint that we have set a too low repetition count, and we need a higher variety of results per steps to get a more consistent course of the

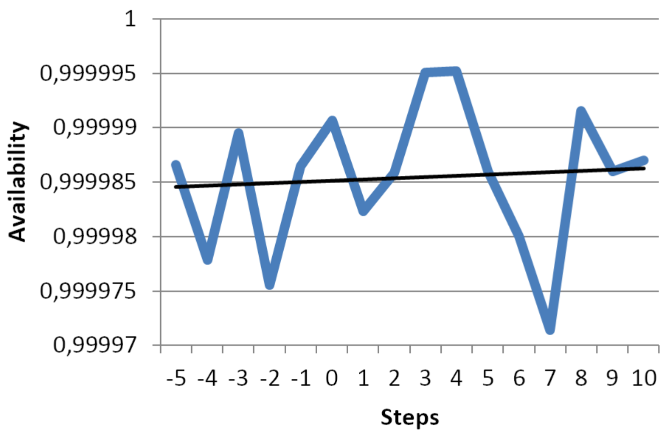


Figure 5.4: Cluster availability (blue) and linear trend line (black) of the *ONOSSWRecRate* parameter steps in the new architecture

steady state availability. Even though, the differences between steps are minimal, they begin to differ from the fifth decimal place onwards. For this reason we do not further investigate this problem.
Despite this, the results trend line matches our expectations as the increasing recovery rate also increases the cluster availability.

In the Figures 6.4, 6.5 and 6.2 we can observe the **hardware, Atomix software and link recovery rate** results. The results match our expectations, as the increasing recovery rate increases the cluster availability in both architectures. Except the hardware recovery rate in the new architecture which stagnates.

5.5.4 Upgrade Rates

For the same reason as in Section 5.5.2 the **upgrade cluster rate** results in Figure 5.5 only consider the new architecture and also it presents a mean of multiple sensitivity analysis as the result is counterintuitive. With a faster upgrade rate the cluster availability decreases slightly. Even though the observed difference from the start of the trend line to its end is minimal[2], we want to give an idea why the trend line could behave this way. With a higher upgrade rate, the cluster has more often instances that are being upgraded, which takes the cluster one instance closer to unavailability. By that we mean, if a cluster has three Atomix instances and one is being upgraded, the cluster is only left

[2]With a value of about 0,0000025 $(0,999985 - 0,9999825)$

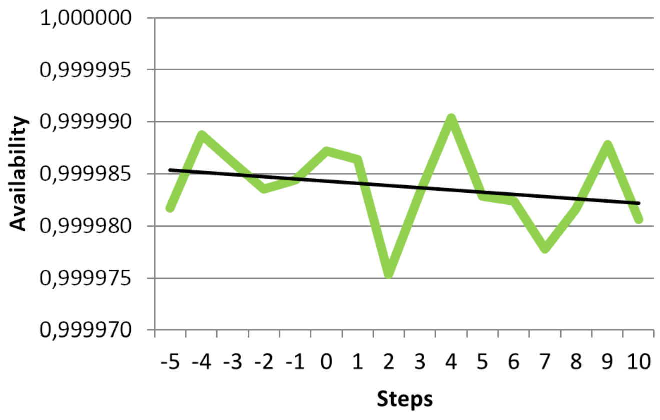


Figure 5.5: Cluster availability (green) and linear trend line (black) of the *UpgradeClusterRate* parameter steps in the new architecture

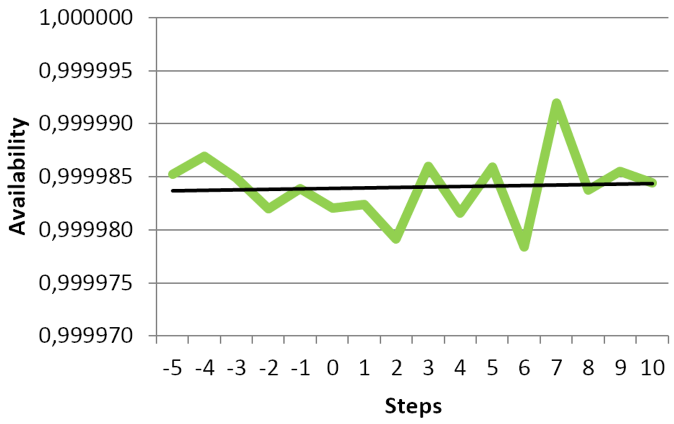


Figure 5.6: Cluster availability (green) and linear trend line (black) of the *ONOSUpgradeFailFinishedRate* parameter steps in the new architecture

with two *up* Atomix instances, if one of them fails, the cluster becomes unavailable as too few *up* Atomix instances are left to keep the consensus *up*.

In contrast to the upgrade rate results before, we can present in Figure 5.6 the results of the **ONOS upgrade finished rate when a problem is encountered** with a slight increase of cluster availability. This is an expected result, as a faster upgrade shortens the time the instance cannot answer requests.

5.6 Chapter Conclusion

We validated our model in various ways and can conclude that our model is valid. In Section 5.1 we showed that the model considers everything it should and does not consider anything it should not. In Sections 5.2 and 5.3 we explained how we created simulations of our model. At last in Sections 5.4 and 5.5 we took a close look at the results of these simulations in terms of sensitivity and plausibility.
Now with a successfully validated model we can begin to evaluate it and answer the research question in Chapter 6.

6 Evaluation

In this chapter we will compare the availability of the two ONOS' architectures. First by taking a new look at the results from the sensitivity analysis from Section 5.4 in Section 6.1. Afterwards we will present specific deployment scenarios and interpret their results in Section 6.2. Finally, in Section 6.3 we will answer the research question, if the architecture change was beneficial for ONOS' availability. It will turn out that the new architecture has an overall higher availability than the old architecture.

6.1 Sensitivity Results interesting for architecture comparison

In the following we will present results of the sensitivity analysis results originally intended for Chapter 5, but they fit much better here as they let us compare the availability in both architectures. These results may not have been gathered if we had executed the simulations manually like mentioned in Section 5.3.

6.1.1 Link fail and recovery rate

In the sensitivity results of links with their fail rate in Figure 6.1 and their recovery rate in Figure 6.2, we can see that the new architecture does not depend on link availability as much as the old architecture does. It also has a higher mean availability throughout the comparison.

We come to this conclusion for the link failure rate since the new architecture trend line changes slightly. The old architecture trend line has a bigger difference between its start and end value. In other words, the higher failure rate is more noticeable in the old architecture.
Additionally, the new architecture trend line stays on a high level between 0,99998 and

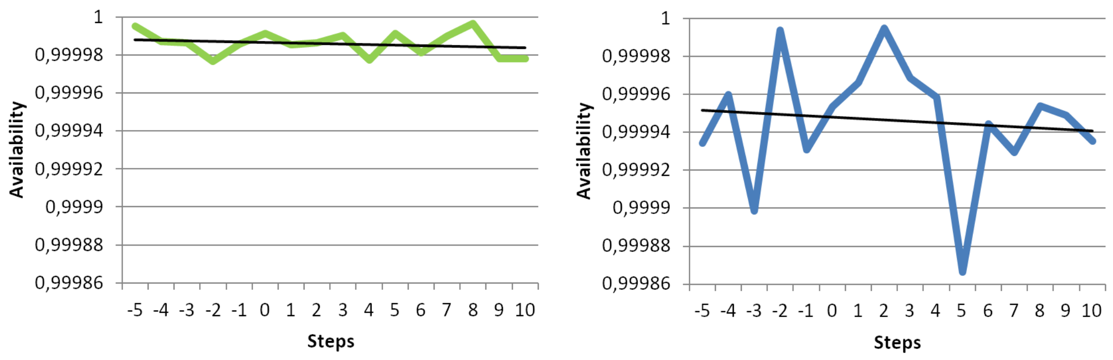


Figure 6.1: Cluster availability (green, blue) and linear trend line (black) of the *LinkFailRate* parameter steps in the new (left) and old (right) architecture

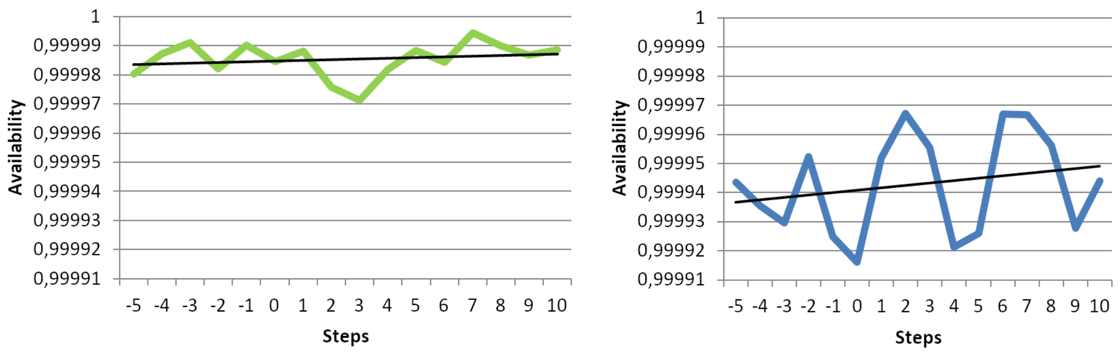


Figure 6.2: Cluster availability (green, blue) and linear trend line (black) of the *LinkRecRate* parameter steps in the new (left) and old (right) architecture

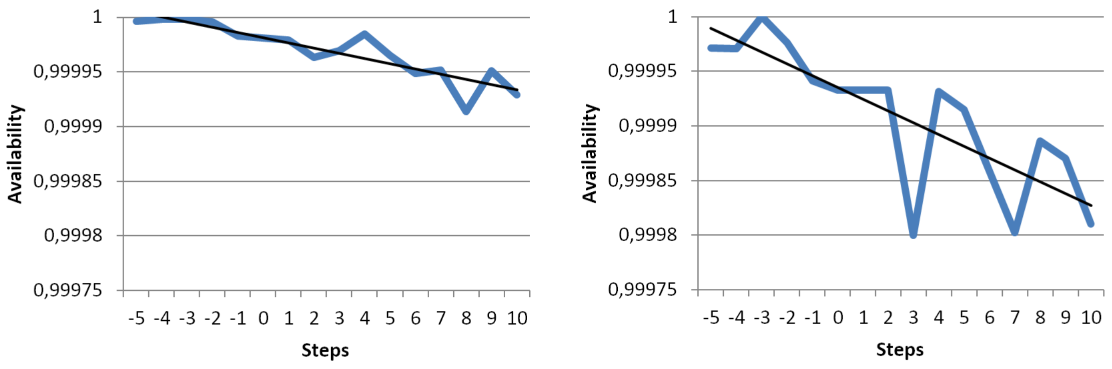


Figure 6.3: Cluster Availability (blue) and linear trend line (black) of the *HWFailRate* parameter steps in the new (left) and old (right) architecture

1. The old architecture comes into that range with two steps, but its trend line is way lower, between 0,99995 and 0,99994.

For the link recovery rate we can observe and interpret the same. The new architecture trend line is on a higher level and also changes slightly, while the old architecture trend line is way lower and has a higher change from the trend line start to end.

At last, one remark to [52]. As described in Section 3.1.3, [52] concludes that link availability is of vital importance to the SDN steady state availability. In the next Section 6.1.2, we will present the hardware rates. When we compare their availability results with the ones of the links, we can see, that in the new architecture the hardware availability has a greater impact than link availability. We reason this as the hardware results are generally on a lower level and their trend lines are steeper. The same could be said about the old architecture, although here the results from the hardware rates and link rates are noticeably closer.

6.1.2 Hardware fail and recovery rate

We can observe in the Figures 6.3 and 6.4 that both architectures behave the same when the hardware failure and recovery rates change. Different is the degree of which the availability is impacted.

The hardware failure rate sensitivity analysis for the new and old architecture in Figure 6.3 clearly shows, that the new architecture is less dependent on its hardware, as its trend line declines less steep when compared to the old architecture.

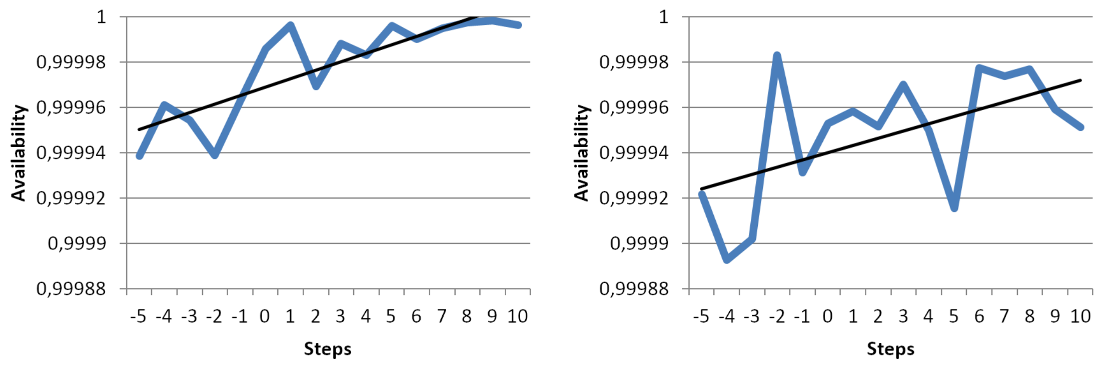


Figure 6.4: Cluster Availability (blue) and linear trend line (black) of the *HWRecRate* parameter steps in the new (left) and old (right) architecture

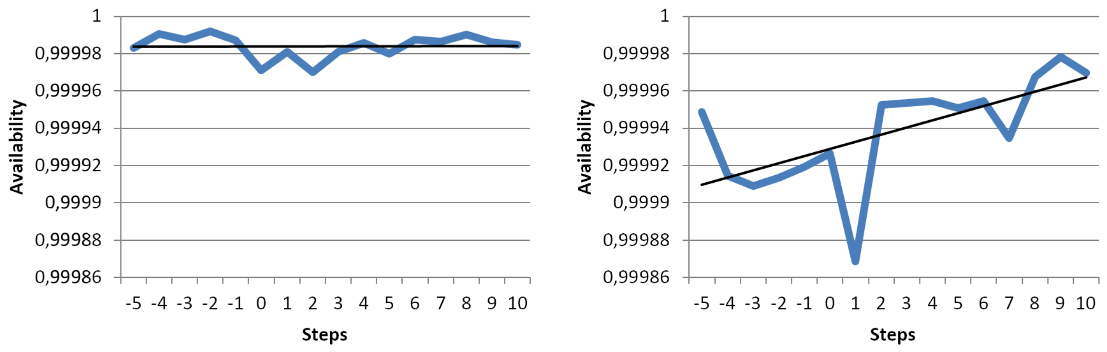


Figure 6.5: Cluster availability (blue) and linear trend line (black) of the *AtomixSWRecRate* parameter steps in the new (left) and old (right) architecture

In the old architecture the hardware recovery rate develops slightly worse than in the new architecture, as with better recovery rates the availability does not rise as fast as in the new architecture. It also does not reach cluster availabilities beyond $0,99998$ as often as the new architecture does, especially in later steps when the recovery rate is 180% (step 8) to 200% (step 10) of its default value.

Like observed in the link rates in Section 6.1.1, the availability of the new architecture is generally higher and less dependent on other elements, compared to the old architecture.

6.1.3 Atomix software recovery rate

In contrast to the examined hardware rates, we can see with the Atomix software recovery rate in Figure 6.5 that the old and new architecture can also behave differently when a parameter changes. In this case this maybe due to the fact that the value of this parameter differs in both architectures. As a reminder, in the new architecture this value represents the Atomix software recovery rate, in the old architecture it represents the ONOS v1.13 software recovery rate.
In the new architecture, the trend line stagnates on a high level, slightly above 0,99998. The old architecture has a steep trend line, so a higher recovery rate increases the cluster availability in the old architecture. Nether the less, even the best availability in the old architecture (roughly 0,999966) is below the average value of the new architecture (slightly above 0,99998). So again we can see what we already observed in Sections 6.1.1 and 6.1.2, that the new architecture generally offers a higher availability than the old architecture.

6.2 Scenarios

Additional to our results from Chapter 5, we now present scenarios to compare the availability in both architectures with specific viewpoints for a more in depth comparison.

6.2.1 Upgrade scenarios

For this deployment scenario, we simulate four scenarios. The two first are the unaltered new and old architecture. The other two are altered new architectures, one version has no upgrade at all, which means that ONOS is never upgraded to improve its availability. The other version of the new architecture considers upgrades, but all ONOS versions have the same availability, so if for example new bugs are introduced with a release, others are fixed. In the following this scenario is called 'stagnating'.

The results of this deployment scenario can be seen in Figure 6.6. In this figure we can observe that the deployment with a stagnating upgrade in the new architecture has the worst availability. Second is the old architecture which is followed up by the unaltered new architecture and the new architecture without any upgrade.

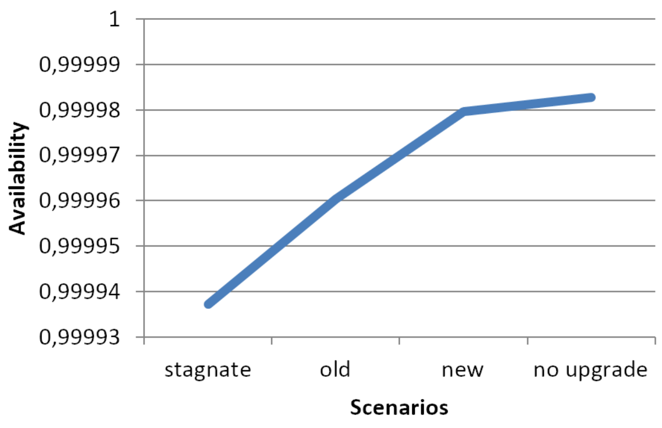


Figure 6.6: Cluster availability of the old and new architectures, including different upgrade behaviours

	ONOS software failure rate decrease		
Scenario	v2	v3	v4
'increase1'	2%	4%	6%
'increase2'	8%	10%	12%
'increase3'	14%	16%	18%
'increase4'	20%	22%	24%

Table 6.1: Decreases of the ONOS software failure rates in the 'increase' scenarios

It makes sense, that the new architecture has a higher availability than the old architecture, as already presented in Sections 5.5.1, 6.1.1, 6.1.2 and 6.1.3. That a new architecture with unchanging failure rates after each upgrade performs the worst can be reasoned with the point that an upgrade always means that during an upgrade of an instance we are one instance closer to unavailability, as described in Section 5.5.4. So in other words we 'buy' the upgrade with availability, but the availability does not improve afterwards to balance out the cost.

We expected that the unaltered new architecture would result in the highest availability, as it contains the improving ONOS availability with each upgrade. Because of this, it is a surprise that the new architecture without any upgrade has the best availability, even though the difference is very low. This surprise maybe due to the fact, that the availability improvement after an upgrade is not high enough to balance out the mentioned cost.

To test if this assumption is true, we simulated the new architecture in four additional scenarios as an extension to the presented scenarios above. Each 'increase' scenario

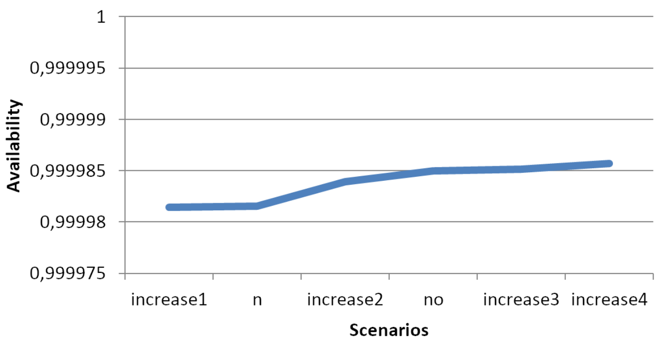


Figure 6.7: Extension of Figure 6.6, Cluster availability of the new architectures, including different upgrade behaviours

increases the ONOS availability, by decreasing the ONOS software failure rate of the upgrades by $(2 + (increasenumber - 1) * 6)\%$. This is done to see, how much the upgrades need to improve the ONOS availability before we can balance out the cost of an upgrade. All decrease percentages are listed in Table 6.1.
The results can be seen in Figure 6.7. We do not further explain the difference between the 'increase1' and the basic new architecture results since their difference is negligible[1]. We can observe, that with increasing ONOS availability, the results come closer to the 'no upgrade' scenario and even can surpass it with 'increase3' and 'increase4'. So if our assumed ONOS software failure rates would decrease significantly of about the value of 'increase3' in Table 6.1 or more, the cluster availability would benefit from upgrades. In other words, the ONOS software failure rate would be needed to decrease around 23% for the first, 33% for the second and 41% for the fourth upgrade when compared to the v1 software failure rate. An overview of the default and needed software failure rates and their decreases in percent can be found within Section A.4 in Table A.4.

6.2.2 Different Atomix and ONOS counts

In this deployment scenario we simulate multiple combinations of Atomix and ONOS instance counts. These combinations are arbitrarily set.
We recommend and do not recommend a certain set of deployment scenarios. The separating line is an availability of 99,99%. This line is arbitrarily set to only recommend deployments that are an improvement over the ONOS availability of 99,99% from [5, 13] mentioned within Section 5.5.1.

[1]0,000000069 $(0,99998152 - 0,9999814151)$

The recommended ones can be seen in Figure 6.8.
Scenarios that deploy the old architecture are named like *<onoscount>'o'* and the new architecture deployments are named like *<onoscount><atomixcount>'n'*. We also will use the term of the 'consensus instances', by which we mean ONOS instances in the old and Atomix instances in the new architecture. We are creating this term as it shorts the explanation, and they respectively are the basis for the consensus status (*up* or *failed*).

The not recommended deployment scenarios have an availability of or below 99,99%. The scenarios *13n* and *19n* have an availability around 99,55%. The reason we assume for the low availability is, that only one ONOS instance needs to fail, to make the cluster unavailable.
We also simulated the scenarios *22n*, *12n* and *2o*, with a result of 0% each. The reason is the realization of the instance partitioning, as described in Section 4.4.1. Only two consensus instances would lead to an immediate partitioned consensus instance, as the immediate transition $PartitionAtomixInstance$ can fire without prior link failure[2]. This could be fixed by altering the model, however, we will not do that since an instance count that is below three or that is even is highly discouraged for high availability deployments [5, 60].

The recommended deployment scenarios in Figure 6.8, have an availability above 99,99%. Interesting to see, is that the consensus instance count mainly influences the cluster availability, to observe between *23n* and *57n* as they are sorted by the second number. With one exception to *29n* which has a very high consensus instance count, but with a very low ONOS count which we assume diminishes the availability, because the cluster is close to unavailability with just two ONOS instances. Especially during an upgrade, with reference to Section 6.2.1.

Although this work only looks at the availability a deployment can offer and does not consider for example load on the instances or operational costs, we want to point out, that an availability beyond 99,999% is achieved in the old architecture with fewer instances than in the new architecture. To reach this availability, the old architectures needs at least five ONOS instances, while the new architecture needs at least three ONOS and five Atomix instances. This result could be interesting for future work, e.g. if this is still true once the model considers load on the instances or the availability is put into context with operational costs.

[2]Because the only input arc preventing the firing has a multiplicity of $alinksub1 = 0$, with reference to our formulas within Figure 4.9

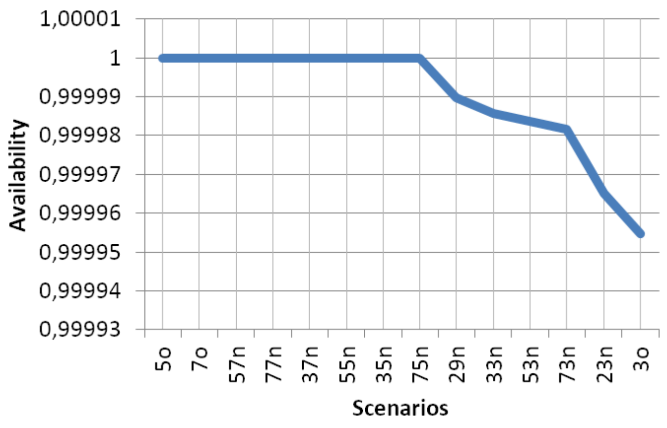


Figure 6.8: Cluster availability of the recommended deployed instance counts

6.3 Evaluation Summary

We presented in Section 6.1.1, that link failures and recoveries do have less impact in the new than in the old architecture. In other words, in environments with unreliable or hard to maintain links, the ONOS instances should be upgraded to at least v1.14 to profit from the higher availability the new architecture offers.
The same is true for the Atomix recovery rate in Section 6.1.3 and the hardware results presented in Section 6.1.2. As here again, the failures and recoveries have less impact onto the new architecture.
The new architecture within ONOS v1.14 or newer, should also be considered in deployments with a low hardware availability. Especially if the hardware's reliability or maintenance is worse than what we assumed in Section 4.3. This is important, as it could mean in this case to lose availability from over 0,99995 to around 0,99875 in our considered worst case for the old architecture (Step 10 in the right chart in Figure 6.3). When looked at in context of a year[3], this means an increase of cluster unavailability of roughly 26 minutes to 657 minutes[4].

That the new architecture performs better in terms of availability than the old architecture can also be seen in our upgrade scenarios in Section 6.2.1 and the deployment scenarios in Section 6.2.2.
In our upgrade scenarios we could observe, that the upgrade comes with an availability cost. So operators of ONOS in the new architecture should not update every release but

[3]Based upon 365 days
[4]Calculated with $(1 - availbility) * 365 * 24 * 60$.

pick the ones that are needed, when viewed from the availability point of view. With reference to [3], one should also wait a few months after the release before upgrading.
When we looked at the instance counts, we could see that for the old architecture at least five ONOS instances should be deployed. In the new architecture the operator should deploy at least five instances of Atomix and at least three instances of ONOS from an availability point of view. In these deployments, both architectures can reach a cluster availability beyond 99,999%. More instances do not drastically increase the cluster's availability in both architectures according to our model.

So, to answer our research question which architecture has a higher cluster steady state availability, we strongly recommend ONOS v1.14 or newer with the new architecture over ONOS v1.13 and older with the old architecture. Especially in environments with worse hardware or link availability than we assumed.
This advantage of the new architecture however does mainly come from the increased count of links and hardware nodes. And so its higher tolerance towards single failures of such elements.

7 Conclusion

In this work we presented and evaluated the research question, if the additional complexity in the new architecture harms the overall availability and if so, if this is only the case in certain scenarios.
This research question was formulated and reasoned in Chapter 4. In this chapter we also defined our model elements and reasoned which are considered and which are not. With these elements specified, we continued to create our GSPN model within GreatSPN.
Our GSPN model was then validated in Chapter 5. We started of with an explanation how the specified elements from Chapter 4 were realized and presented arguments why this was done correctly. We then continued with the explanation of the manual simulation, used during model development, and automated simulations, used for the following sensitivity and plausibility analysis. In the first we could show, that no parameter does overwhelmingly influence the model results. In the latter we presented further results, explaining if these were expected or not. Although some results were counterintuitive, they could still be reasoned and are plausible.
With a validated model we continued to Chapter 6. We took once more a look at the results of simulations and could observe that the new architecture performs better than the old architecture in terms of availability. This was confirmed with specific upgrade and deployment scenarios. The chapter was concluded with, besides others, recommendations for ONOS deployments and the overall recommendation to deploy the new ONOS architecture, this is also a signal to the ONOS developers that their architecture change was beneficial for ONOS' availability. Even though, the fact that the new architecture can be easily upgraded, was not the main factor for the availability gain over the old architecture in our model. This was foreshadowed in the counterintuitive results in Chapter 6 mentioned above. The availability gain over the old architecture is based upon the addition of new hardware nodes and especially links. Single hardware or link failures can be much easier tolerated as the cluster has a higher replication degree.
We conclude this work, with a clear recommendation for the new ONOS architecture in ONOS version 1.14 and onwards.

Our work can be extended in multiple ways.

One idea is to create a more realistic model, e.g. include load dependent reliability into the model or include a more sophisticated network partitioning. Additional ideas can be found in Sections 4.2.2 and 6.2.2.
Also, we mentioned in the introduction, that the new architecture can be used for easier horizontal scaling. This could be combined with a model that includes load dependent reliability as mentioned above.
Here we want to remind the reader, that some of our rates, as explained in Section 4.3, are based upon older literature or are guessed. So our results should be taken with a grain of salt. Further specific and reliable rates are needed for future work for more realistic performance analyzations.
The mentioned ideas could all lead to more realistic results that make more reliable or fine-grained recommendations possible.

As mentioned in Section 4.4, we thought about moving to SWNs, when the model becomes more complex this may help to improve readability and maintainability of the model. Additionally, one could think about creating a hierarchical model like done in [51, 52] as mentioned in Chapter 3.
These ideas might be needed if the model becomes more complex, like with the mentioned ideas above. A SWN or hierarchical model might in this case be easier to read, maintain and extend for even further future work.

Bibliography

[1] : *ONOS Security and Performance Analysis.* Dezember 2017. – URL https://opennetworking.org/wp-content/uploads/2017/07/ONOS-security-and-performance-analysis-brigade-report-no1.pdf

[2] : *ONOS Security and Performance Analysis (Report No. 2).* November 2018. – URL https://opennetworking.org/wp-content/uploads/2018/11/secperf_report_2.pdf

[3] : *Assessing the Maturity of SDN Controllers with Reliability Growth models.* Juni 2019. – URL https://wiki.onosproject.org/download/attachments/12422167/tum-onfworkshop.pdf?version=1&modificationDate=1561629548175&api=v2

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

[5] : *ONOS Confluence.* October 2020. – URL https://wiki.onosproject.org/display/ONOS/ONOS. – Zugriffsdatum: 16.10.2020

[6] Alam, Iqbal ; Sharif, Kashif ; Li, Fan ; Latif, Zohaib ; Karim, M. M. ; Biswas, Sujit ; Nour, Boubakr ; Wang, Yu: A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV. In: *ACM Comput. Surv.* 53 (2020), April, Nr. 2. – URL https://doi.org/10.1145/3379444. – ISSN 0360-0300

[7] Ardagna, Danilo ; Barbierato, Enrico ; Evangelinou, Athanasia ; Gianniti, Eugenio ; Gribaudo, Marco ; Pinto, Túlio B. M. ; Guimarães, Anna ; Silva, Ana P. Couto da ; Almeida, Jussara M.: Performance Prediction of Cloud-Based Big Data Applications. In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering.* New York, NY, USA : Association for Computing Machinery, 2018 (ICPE '18), S. 192–199. – URL https://doi.org/10.1145/3184407.3184420. – ISBN 9781450350952

[8] Avizienis, A. ; Laprie, J. .. ; Randell, B. ; Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), Nr. 1, S. 11–33

[9] Avizienis, Algirdas ; Laprie, Jean-Claude ; Randell, Brian ; Landwehr, Carl: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Trans. Dependable Secur. Comput.* 1 (2004), Januar, Nr. 1, S. 11–33. – URL http://dx.doi.org/10.1109/TDSC.2004.2. – ISSN 1545-5971

[10] Baarir, Soheib ; Beccuti, Marco ; Cerotti, Davide ; De Pierro, Massimiliano ; Donatelli, Susanna ; Franceschinis, Giuliana: The GreatSPN Tool: Recent Enhancements. In: *SIGMETRICS Perform. Eval. Rev.* 36 (2009), März, Nr. 4, S. 4–9. – URL https://doi.org/10.1145/1530873.1530876. – ISSN 0163-5999

[11] Ballarini, P. ; Bernardi, S. ; Donatelli, S.: Validation and evaluation of a software solution for fault tolerant distributed synchronization. In: *Proceedings International Conference on Dependable Systems and Networks*, 2002, S. 773–782

[12] Benzekki, Kamal ; El Fergougui, Abdeslam ; Elbelrhiti Elalaoui, Abdelbaki: Software-defined networking (SDN): a survey. In: *Security and Communication Networks* 9 (2016), Nr. 18, S. 5803–5833. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737

[13] Berde, Pankaj ; Gerola, Matteo ; Hart, Jonathan ; Higuchi, Yuta ; Kobayashi, Masayoshi ; Koide, Toshio ; Lantz, Bob ; O'Connor, Brian ; Radoslavov, Pavlin ; Snow, William ; Parulkar, Guru: ONOS: Towards an Open, Distributed SDN OS. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA : Association for Computing Machinery, 2014 (HotSDN '14), S. 1–6. – URL https://doi.org/10.1145/2620728.2620744. – ISBN 9781450329897

[14] Brosch, Franz ; Buhnova, Barbora ; Koziolek, Heiko ; Reussner, Ralf: Reliability Prediction for Fault-Tolerant Software Architectures. New York, NY, USA : Association for Computing Machinery, 2011 (QoSA-ISARCS '11), S. 75–84. – URL https://doi.org/10.1145/2000259.2000274. – ISBN 9781450307246

[15] Cachin, Christian ; Guerraoui, Rachid ; Rodrigues, Luis: *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011. – URL http://infoscience.epfl.ch/record/208942

[16] Canini, M. ; Salem, I. ; Schiff, L. ; Schiller, E. M. ; Schmid, S.: Renaissance: A Self-Stabilizing Distributed SDN Control Plane. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, S. 233–243

[17] Casale, Giuliano ; Muntz, Richard R. ; Serazzi, Giuseppe: Special Issue on Tools for Computer Performance Modeling and Reliability Analysis. In: *SIGMETRICS Perform. Eval. Rev.* 36 (2009), März, Nr. 4, S. 2–3. – URL https://doi.org/10.1145/1530873.1530875. – ISSN 0163-5999

[18] Caselli, S. ; Conte, G.: GSPN models of concurrent architectures with mesh topology. In: *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*. Los Alamitos, CA, USA : IEEE Computer Society, dec 1991, S. 280,281,282,283,284,285,286,287,288,289. – URL https://doi.ieeecomputersociety.org/10.1109/PNPM.1991.238792

[19] Chang, Michael A. ; Tschaen, Bredan ; Benson, Theophilus ; Vanbever, Laurent: Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. New York, NY, USA : Association for Computing Machinery, 2015 (SIGCOMM '15), S. 371–372. – URL https://doi.org/10.1145/2785956.2790038. – ISBN 9781450335423

[20] Cox, J. H. ; Chung, J. ; Donovan, S. ; Ivey, J. ; Clark, R. J. ; Riley, G. ; Owen, H. L.: Advancing Software-Defined Networks: A Survey. In: *IEEE Access* 5 (2017), S. 25487–25526

[21] Curic, Maja ; Carle, Georg ; Despotovic, Zoran ; Khalili, Ramin ; Hecker, Artur: SDN on ACIDs. In: *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*. New York, NY, USA : Association for Computing Machinery, 2017 (CAN '17), S. 19–24. – URL https://doi.org/10.1145/3155921.3155924. – ISBN 9781450354233

[22] Diallo, O. ; Sene, M. ; Sarr, I.: Freshness-aware metadata management: Performance evaluation with SWN models. In: *ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*, 2010, S. 1–6

[23] Dobrijevic, Ognjen ; Natalino, Carlos ; Furdek, Marija ; Hodzic, Haris ; Dzanko, Matija ; Wosinska, Lena: Another Price to Pay: An Availability Analysis for SDN Virtualization with Network Hypervisors, 08 2018

[24] Farr, William: Software reliability modeling survey. In: *Handbook of software reliability engineering* (1996), S. 71–117

[25] Gokhale, S. S. ; Trivedi, K. S.: Analytical Models for Architecture-Based Software Reliability Prediction: A Unification Framework. In: *IEEE Transactions on Reliability* 55 (2006), Nr. 4, S. 578–590

[26] Gokhale, Swapna S.: Architecture-based software reliability analysis: Overview and limitations. In: *IEEE Transactions on dependable and secure computing* 4 (2007), Nr. 1, S. 32–40

[27] Goseva-Popstojanova, K. ; Mathur, A. P. ; Trivedi, K. S.: Comparison of architecture-based software reliability models. In: *Proceedings 12th International Symposium on Software Reliability Engineering*, 2001, S. 22–31

[28] Goševa-Popstojanova, Katerina ; Trivedi, Kishor S.: Architecture-based approach to reliability assessment of software systems. In: *Performance Evaluation* 45 (2001), Nr. 2-3, S. 179–204

[29] Gray, J. ; Siewiorek, D. P.: High-availability computer systems. In: *Computer* 24 (1991), Nr. 9, S. 39–48

[30] Guan, X. ; Choi, B. ; Song, S.: Reliability and Scalability Issues in Software Defined Network Frameworks. In: *2013 Second GENI Research and Educational Experiment Workshop*, March 2013, S. 102–103

[31] Hanmer, R. ; Jagadeesan, L. ; Mendiratta, V. ; Zhang, H.: Friend or Foe: Strong Consistency vs. Overload in High-Availability Distributed Systems and SDN. In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, S. 59–64

[32] Hanmer, R. ; Liu, S. ; Jagadeesan, L. ; Rahman, M. R.: Death by Babble: Security and Fault Tolerance of Distributed Consensus in High-Availability Softwarized Networks. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, S. 266–270

[33] Harrison, Neil B. ; Avgeriou, Paris: Incorporating Fault Tolerance Tactics in Software Architecture Patterns. In: *Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*. New York, NY, USA : Association for Computing Machinery, 2008 (SERENE '08), S. 9–18. – URL https://doi.org/10.1145/1479772.1479775. – ISBN 9781605582757

[34] HARRISON, Neil B. ; AVGERIOU, Paris ; ZDUN, Uwe: On the Impact of Fault Tolerance Tactics on Architecture Patterns. In: *Proceedings of the 2nd International Workshop on Software Engineering for Resilient Systems*. New York, NY, USA : Association for Computing Machinery, 2010 (SERENE '10), S. 12–21. – URL https://doi.org/10.1145/2401736.2401738. – ISBN 9781450302890

[35] HERKER, S. ; AN, X. ; KIESS, W. ; BEKER, S. ; KIRSTAEDTER, A.: Data-Center Architecture Impacts on Virtualized Network Functions Service Chain Embedding with High Availability Requirements. In: *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015, S. 1–7

[36] IMMONEN, Anne ; NIEMELÄ, Eila: Survey of reliability and availability prediction methods from the viewpoint of software architecture. In: *Software & Systems Modeling* 7 (2008), Nr. 1, S. 49

[37] JOHNSON, Allen M. ; MALEK, Miroslaw: Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability. In: *ACM Comput. Surv.* 20 (1988), Dezember, Nr. 4, S. 227–269. – URL https://doi.org/10.1145/50020.50062. – ISSN 0360-0300

[38] KLEPPMANN, M.: *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017. – ISBN 9781491903100

[39] KOBO, Hlabishi I. ; ABU-MAHFOUZ, Adnan M. ; HANCKE, Gerhard P.: Efficient controller placement and reelection mechanism in distributed control system for software defined wireless sensor networks. In: *Transactions on Emerging Telecommunications Technologies* 30 (2019), Nr. 6, S. e3588. – URL https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3588. – e3588 ett.3588

[40] KRIAA, S. ; PAPILLON, S. ; JAGADEESAN, L. ; MENDIRATTA, V.: Better Safe than Sorry: Modeling Reliability and Security in Replicated SDN Controllers. In: *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, 2020, S. 1–6

[41] LAI, R.: Performance modelling for the CSMA/CD protocol using GSPN. In: *Proceedings of IEEE Singapore International Conference on Networks and International Conference on Information Engineering '95*, 1995, S. 126–130

[42] LAPRIE, J. .. ; ARLAT, J. ; BEOUNES, C. ; KANOUN, K.: Definition and analysis of hardware- and software-fault-tolerant architectures. In: *Computer* 23 (1990), Nr. 7, S. 39–51

[43] LI, T. ; LI, Y. ; QIAN, Y. ; XU, Y.: Optimizing reliability, maintainability and testability parameters of equipment based on GSPN. In: *Journal of Systems Engineering and Electronics* 26 (2015), Nr. 3, S. 633–643

[44] LIU, H. ; WANG, L. ; LI, Z. ; HU, Y.: Improving Risk Evaluation in FMEA With Cloud Model and Hierarchical TOPSIS Method. In: *IEEE Transactions on Fuzzy Systems* 27 (2019), Nr. 1, S. 84–95

[45] LONGO, Francesco ; DISTEFANO, Salvatore ; BRUNEO, Dario ; SCARPA, Marco: Dependability modeling of Software Defined Networking. In: *Computer Networks* 83 (2015), S. 280 – 296. – URL http://www.sciencedirect.com/science/article/pii/S1389128615001139. – ISSN 1389-1286

[46] LÓPEZ-GRAO, Juan P. ; MERSEGUER, José ; CAMPOS, Javier: On the use of formal models in Software Performance Evaluation. In: *Actas de las X Jornadas de Concurrencia* (2002), S. 367–387

[47] LÓPEZ-GRAO, Juan P. ; MERSEGUER, José ; CAMPOS, Javier: From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. In: *SIGSOFT Softw. Eng. Notes* 29 (2004), Januar, Nr. 1, S. 25–36. – URL https://doi.org/10.1145/974043.974048. – ISSN 0163-5948

[48] MARSAN, Marco A. ; BALBO, G. ; CONTE, Gianni ; DONATELLI, S. ; FRANCESCHINIS, G.: *Modelling with Generalized Stochastic Petri Nets*. 1st. USA : John Wiley & Sons, Inc., 1994. – ISBN 0471930598

[49] MARWAH, Manish ; MACIEL, Paulo ; SHAH, Amip ; SHARMA, Ratnesh ; CHRISTIAN, Tom ; ALMEIDA, Virgilio ; ARAÚJO, Carlos ; SOUZA, Erica ; CALLOU, Gustavo ; SILVA, Bruno ; GALDINO, Sérgio ; PIRES, Jose: Quantifying the Sustainability Impact of Data Center Availability. In: *SIGMETRICS Perform. Eval. Rev.* 37 (2010), März, Nr. 4, S. 64–68. – URL https://doi.org/10.1145/1773394.1773405. – ISSN 0163-5999

[50] MENDIRATTA, V. B. ; JAGADEESAN, L. J. ; HANMER, R. ; RAHMAN, M. R.: How Reliable Is My Software-Defined Network? Models and Failure Impacts. In: *2018*

IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2018, S. 83–88

[51] Nencioni, G. ; Helvik, B. E. ; Gonzalez, A. J. ; Heegaard, P. E. ; Kamisinski, A.: Availability Modelling of Software-Defined Backbone Networks. In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, S. 105–112

[52] Nguyen, T. A. ; Eom, T. ; An, S. ; Park, J. S. ; Hong, J. B. ; Kim, D. S.: Availability Modeling and Analysis for Software Defined Networks. In: *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2015, S. 159–168

[53] Owre, S. ; Rushby, J. ; Shankar, N. ; von Henke, F.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 2, S. 107–125

[54] Sakic, E. ; Kellerer, W.: Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane. In: *IEEE Transactions on Network and Service Management* 15 (2018), Nr. 1, S. 304–318

[55] Sanghare, O. A. ; Sene, M. ; Rodrigues, J. J. P. C.: Distributed Transactions on Mobile Systems: Performance Evaluation Using SWN. In: *2011 IEEE International Conference on Communications (ICC)*, 2011, S. 1–6

[56] Santos, Guto L. ; Endo, Patricia T. ; Gonçalves, Glauco ; Rosendo, Daniel ; Gomes, Demis ; Kelner, Judith ; Sadok, Djamel ; Mahloo, Mozhgan: Analyzing the it subsystem failure impact on availability of cloud services. In: *2017 IEEE symposium on computers and communications (ISCC)* IEEE (Veranst.), 2017, S. 717–723

[57] Scott-Hayward, Sandra: Design and deployment of secure, robust, and resilient SDN Controllers. In: *2015 1st IEEE Conference on Network Softwarization (NetSoft)*, Institute of Electrical and Electronics Engineers (IEEE), April 2015. – IEEE Conference on Network Softwarization (NetSoft 2015) ; Conference date: 13-04-2015 Through 17-04-2015. – ISBN 978-1-4799-7898-4

[58] Sheoran, Amit ; Xiangyu, Bu ; Lianjie, Cao ; Puneet, Sharma ; Sonia, Fahmy: An Empirical Case for Container-driven Fine-grained VNF Resource Flexing, 2016

[59] SIDKI, L. ; BEN-SHIMOL, Y. ; SADOVSKI, A.: Fault tolerant mechanisms for SDN controllers. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, S. 173–178

[60] TANENBAUM, Andrew S. ; STEEN, Maarten v.: *Verteilte Systeme - Prinzipien und Paradigmen*. Pearson Studium, 2008. – ISBN 978-3-827-37293-2

[61] TANG, Y. ; WU, Y. ; CHENG, G. ; XU, Z.: Intelligence Enabled SDN Fault Localization via Programmable In-band Network Telemetry. In: *2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, 2019, S. 1–6

[62] TEKINERDOGAN, Bedir ; SOZER, Hasan ; AKSIT, Mehmet: Software architecture reliability analysis using failure scenarios. In: *Journal of Systems and Software* 81 (2008), Nr. 4, S. 558 – 575. – URL http://www.sciencedirect.com/science/article/pii/S0164121207003032. – Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006). – ISSN 0164-1212

[63] THIMMARAJU, Kashyap ; SHASTRY, Bhargava ; FIEBIG, Tobias ; HETZELT, Felicitas ; SEIFERT, Jean-Pierre ; FELDMANN, Anja ; SCHMID, Stefan: Taking Control of SDN-Based Cloud Systems via the Data Plane. In: *Proceedings of the Symposium on SDN Research*. New York, NY, USA : Association for Computing Machinery, 2018 (SOSR '18). – URL https://doi.org/10.1145/3185467.3185468. – ISBN 9781450356640

[64] TSAREV, R. Y. ; GRUZENKIN, D. V. ; KOVALEV, I. V. ; PROKOPENKO, A. V. ; KNYAZKOV, A. N.: Evaluation of availability of cluster distributed disaster tolerant systems for control and information processing based on a cluster quorum. In: *IOP Conference Series: Materials Science and Engineering* Bd. 155, 2016

[65] VILCHEZ, J. M. S. ; SARMIENTO, D. E.: Fault Tolerance Comparison of ONOS and OpenDaylight SDN Controllers. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, S. 277–282

[66] VIZARRETA, P. ; HEEGAARD, P. ; HELVIK, B. ; KELLERER, W. ; MACHUCA, C. M.: Characterization of failure dynamics in SDN controllers. In: *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*, 2017, S. 1–7

[67] Vizarreta, P. ; Trivedi, K. ; Mendiratta, V. ; Kellerer, W. ; Mas-Machuca, C.: DASON: Dependability Assessment Framework for Imperfect Distributed SDN Implementations. In: *IEEE Transactions on Network and Service Management* 17 (2020), Nr. 2, S. 652–667

[68] Wang, Shaoping ; Tang, Daoyu: Availability analysis based on performance for LAN. In: Zhang, Guangjun (Hrsg.) ; Zhao, Huijie (Hrsg.) ; Wang, Zhongyu (Hrsg.): *Fifth International Symposium on Instrumentation and Control Technology* Bd. 5253 International Society for Optics and Photonics (Veranst.), SPIE, 2003, S. 972 – 976. – URL https://doi.org/10.1117/12.522336

[69] Zhang, Hao ; Lu, Minyan ; Gu, Tingyang: A Petri-Net Based Reliability Prediction Method for SOA Software. In: *Proceedings of the 2nd International Conference on Advances in Image Processing.* New York, NY, USA : Association for Computing Machinery, 2018 (ICAIP '18), S. 165–172. – URL https://doi.org/10.1145/3239576.3239606. – ISBN 9781450364607

[70] Zhang, Yang ; Ramadan, Eman ; Mekky, Hesham ; Zhang, Zhi-Li: When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network. In: *Proceedings of the First Asia-Pacific Workshop on Networking.* New York, NY, USA : Association for Computing Machinery, 2017 (APNet'17), S. 1–7. – URL https://doi.org/10.1145/3106989.3106999. – ISBN 9781450352444

[71] Zhong, Duhang ; Qi, Zhichang: A Petri Net Based Approach for Reliability Prediction of Web Services. In: Meersman, Robert (Hrsg.) ; Tari, Zahir (Hrsg.) ; Herrero, Pilar (Hrsg.): *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, S. 116–125. – ISBN 978-3-540-48272-7

A Appendix

In the appendix we present additional information that may interest readers. We explain further information of the stacked failure extension in Section A.1, the used tool to start simulations & gather their results in Section A.2, the full URLs for the used shortened links in Section A.3, ONOS software failure rates to surpass the 'no upgrade' scenario from Section 6.2.1 in Section A.4, and the full GSPN model in Section A.5.

A.1 Stacked Failure Extension

To model stacked failures, we need to consider the dependencies the have between each other. For a start, any failure (software, hardware, network partition) can happen on top of any other, with one exception. Software failures are overwritten by a hardware failure, as a software cannot be executed on top of failed hardware. This can be seen in Figure A.1. To improve readability, the failure transitions are horizontal and the recovery transitions are immediate and vertical. This has no semantic meaning.

This becomes more complex if we want to incorporate that into the presented model, we need to keep track which instance has currently which failures, to make sure it is recovered the correct way. For example a partitioned node with a software failure, must recover both before that instance is *up* again. Especially for ONOS, this would drastically increase the amount of transitions and places, as we have to model each specific failure scenario for each version.

A.2 Simulation Tool

The tool mentioned in Chapter 5 consists of BASH and Python code.
BASH scripts start and handle the simulations, while the Python scripts are used to

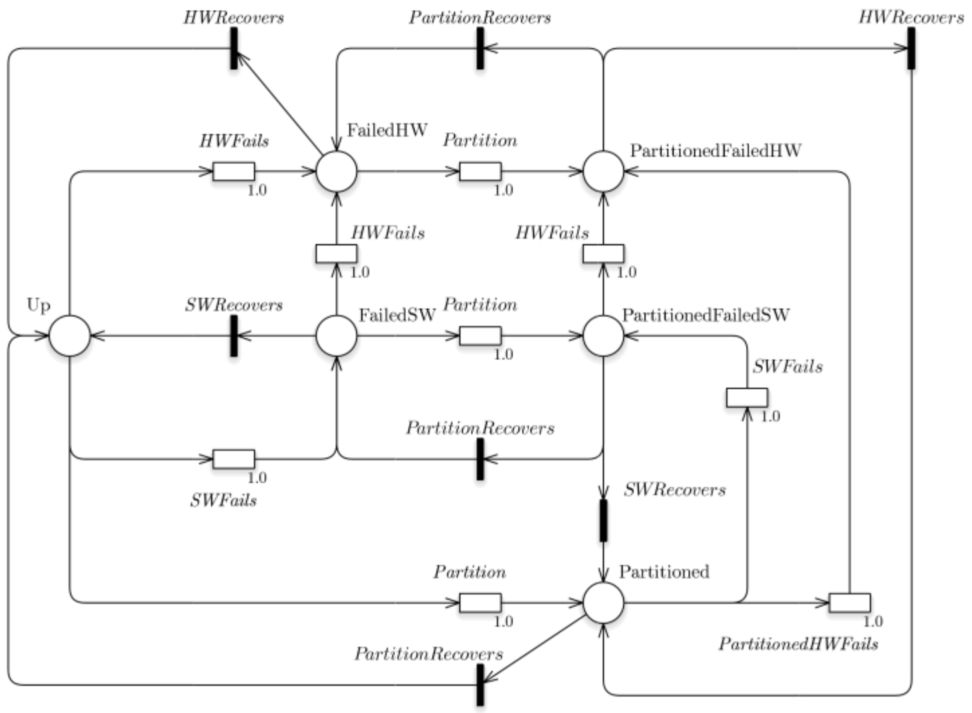


Figure A.1: Basic idea for stacked failure extension

gather results and write the *xls* file. Initially the result gathering was also done in BASH, we quickly re-implemented the logic in Python as it gave us an immense performance advantage, for example to parse a 10.000 line text file BASH needed several minutes, Python can do that in matter of seconds. This is even more helpful once these text files reach 54 GB in combined size.

For the *xls* file creation we use the *xlsxwriter* library which is very useful. With that we even can create line charts in a few lines of code. The manual creation of these charts, as we have done in previous projects, is a tedious and error prone task we are glad to have getting rid of by automation.

In Listing A.1 we show an example configuration of the simulation tool. With that configuration we have a total of 320 simulations[1]. Interesting point is that not every parameter is analyzed in both architectures like in the example configuration, e.g. *UpgradeClusterRate* is not analyzed in the old architecture as there is no upgrading and so it has no impact on its availability, in that case *architecturecount* would be 1. As a side note, the count of concurrent simulations should not exceed the thread count as each simulation fully utilizes one thread, in our case we choose 9 with a 12 thread CPU.

```
approximation=50 # Markov Steady State Approximation in percent
min=-5           # Minimum of steps from default value
max=10           # Maximum of steps from default value
repetitions=5    # Count of simulations repetion, each with own seed
step_diff_percent=10 # Difference per step to the default value in percent
sleep_between_starts=0.1s # Wait time between parallel simulation starts
simultanious_containers=9 # Maximum number of concurrent simulations
timeout_min=60            # Timeout per simulation
params='HWFailRate HWRecRate' # Analyzed Parameters
```

Listing A.1: Example simulation tool configuration

Furthermore, the tool is split in the following files to make the code more readable:

1. *clean_ results.sh*: Deletes the results and temporary simulation files to have a clean setup
2. *create_ comparison.sh*: The tool's entry point

[1]Calculated with $stepcount * repetitions * analyzedparameters * architecturecount = 16 * 5 * 2 * 2$

3. *create_csvs.sh*: Gathers the simulation results and puts them into *csv* files with the help of *get_mean_from_wnsimlog.py*
4. *create_xls.py*: Creates one *xls* file from the *csv* files
5. *get_mean_from_wnsimlog.py*: Gather the availability out of the result files, this is the mentioned task that is much quicker with Python
6. *start_sim.sh*: Start a simulation with a given set of parameters in sequence or parallel, also keeps track of running simulations and executes the timeout
7. *util_container.sh*: Functions to interact with the containers
8. *util_log.sh*: Functions for logging
9. *util_math.sh*: Functions for easier calculations within BASH
10. *util_sim_params.sh*: Configuration of the simulation tools, Listing A.1 would be defined here.

The simulation tool can be requested and may be uploaded to GitHub in the future.

A.3 URL Links

We used shortened links for long URLs. It could be, that after some time, the shortened URLs break. For this reason we provide the full URLs here:

```
https://tinyurl.com/u39fj9px = https://groups.google.com/a/onosproject.org/g/
    onos-dev/c/iu_iP8pFs-U/m/OGtYzVy_CwAJ
https://tinyurl.com/2n8u4jfp = https://groups.google.com/a/onosproject.org/g/
    onos-dev/c/hzfjjEyruGo/m/xsnEiMCdAwAJ
https://tinyurl.com/2yb86t8b = https://docs.google.com/document/d/1-
    xZ3Wnr6VZS34paYVdZhF8WWo3M6VWKNJXQTmBnnM7Y
https://tinyurl.com/fhkzmvae = https://atomix.io/docs/latest/user-manual/
    deployment/kubernetes/
```

Listing A.2: Full URLs of shortened links

A.4 ONOS Software failure rates to surpass 'no upgrade' scenario

With reference to Section 6.2.1, we now list the needed ONOS software failure rate for each version, so our model could benefit from upgrades in terms of availability. These numbers are based upon our parameters described in Section 4.3.

	Default		To surpass 'no upgrade' scenario	
Version	Rate	Decrease	Rate	Decrease
v1	2,1067	0%	2,1067	0%
v2	1,8725	11,12%	1,6104	23,56%
v3	1,6750	20,49%	1,4070	33,21%
v4	1,5042	28,60%	1,2334	41,45%

Table A.1: Decreases of the ONOS software failure rates to surpass the 'no upgrade' scenario

A.5 Full GSPN model

In Figure A.2 we added the full GSPN model, in addition to the shown squares in Section 4.4, so that one can better understand how the squares are connected.

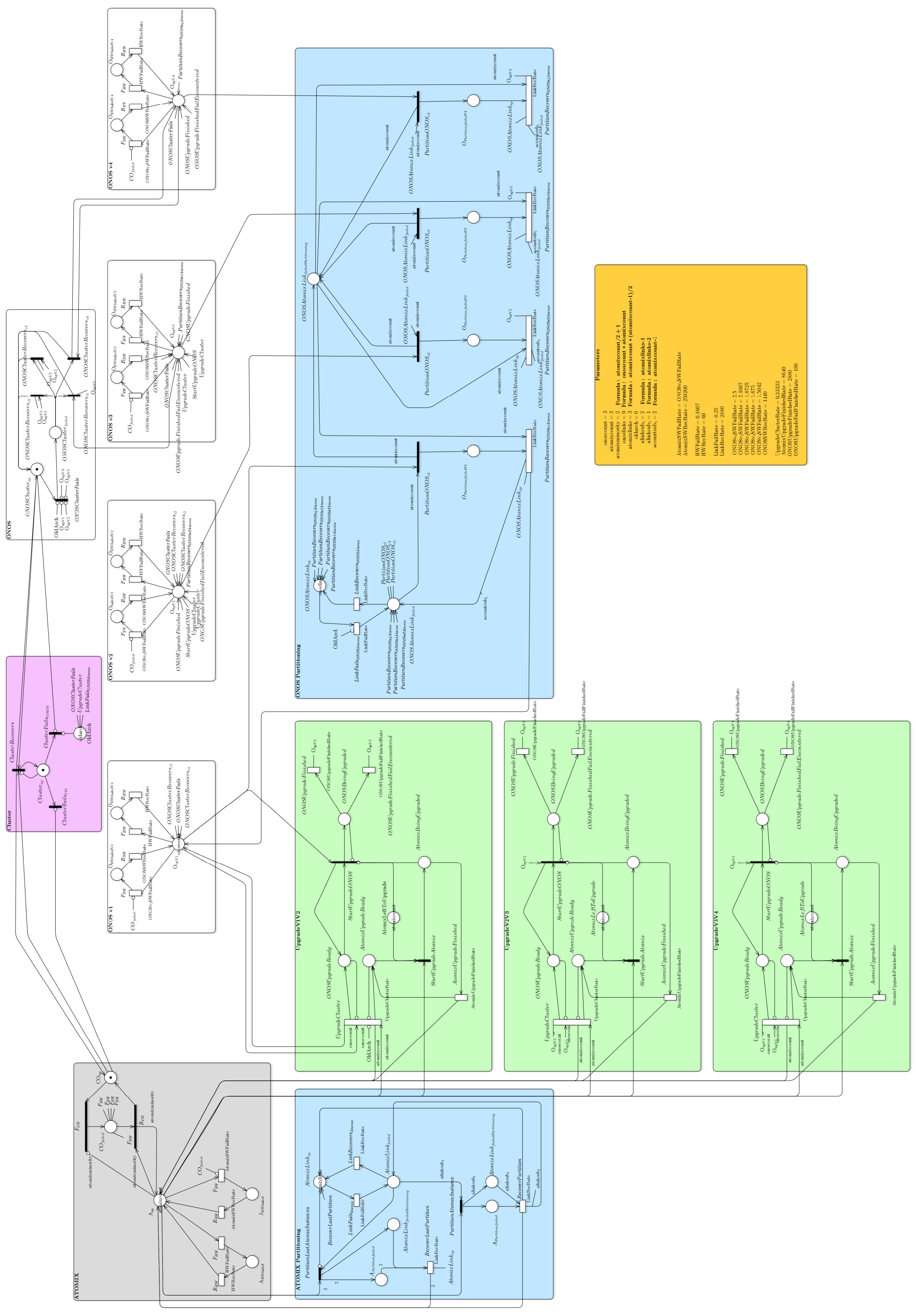


Figure A.2: Our model in GSPN and GreatSPN

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name:	______________________
Vorname:	______________________

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Verfügbarkeitsanalyse der ONOS Architektur

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	______________________
Ort	Datum	Unterschrift im Original