Model-Driven Availability Assessment of the NFV-MANO with Software Rejuvenation

Besmir Tola, Yuming Jiang, and Bjarne E. Helvik

***Abstract*—Network Function Virtualization enables network operators to modernize their networks with greater elasticity, network programmability, and scalability. Exploiting these advantages requires new and specialized designs for management, automation, and orchestration systems which are capable of reliably operating and handling new elements such as virtual functions, virtualized infrastructures, and a whole new set of relationships among them. Operations such as resource allocation, instantiation, monitoring, scaling, or termination of virtual functions are key lifecycle operations that NFV management and orchestration (NFV-MANO) frameworks need to correctly perform. Failures of the NFV-MANO prevent the network ability to respond to new service requests or events related to the normal lifecycle operation of network services. Thus, it is important to ensure robustness and high availability of the MANO framework. This paper adopts a model-driven approach to predict the availability of the NFV-MANO and assess the impact that different failure modes have. We propose different models, based on Stochastic Activity Networks (SANs), which abstract various MANO deployment configurations, inspired by current containerized open-source MANO implementations. Moreover, we integrate software rejuvenation and investigate the trade-off between its associated overhead and system availability increase. An extensive experimental campaign with fault-injection techniques on a real-life MANO implementation allows to derive a number of realistic recovery parameters. The case studies are used to quantitatively evaluate the steady-state availability and identify the most important parameters influencing system availability for the different deployment configurations.**

***Index Terms*—NFV-MANO, Availability, Software aging, Software rejuvenation, SAN models, Containers.**

I. Introduction

NETWORK Function Virtualization (NFV) empowers an innovative transformation of today's network architectures. At the core of the paradigm lies the separation of the network functions from the underlying hardware platforms. Network-based services can be realized through virtualized software entities, commonly referred to as Virtualized Network Functions (VNFs), which can be executed in general purpose hardware rather than requiring specialized purpose-built platforms. They can embody network functions such as Routers (vRouter), Firewalls (vFW), and Load Balancers (vLB) [1], and can be chained together to provide advanced full-scale network services [2], [3].

As defined by the European Telecommunications Standards Institute (ETSI), the standard high-level architecture of NFV incorporates three main blocks that are the NFV infrastructure (NFVI), the VNFs, and a logically centralized Management and Orchestration (MANO) entity [4]. The NFVI provides a virtualization environment for the deployment and execution of VNFs, including virtual compute, storage and networking resources. VNFs are software implementations of network functions which should be able to interact with other VNFs for providing composed network services. The MANO performs life-cycle management of VNFs and NFs, and the orchestration of infrastructure resources supporting their execution.

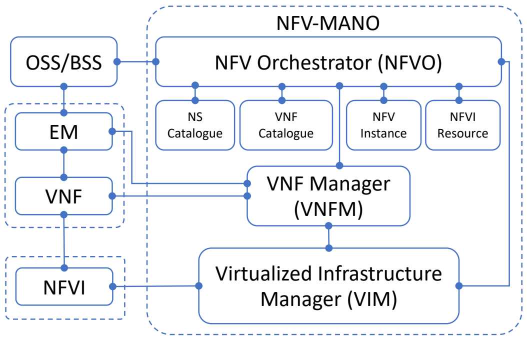


Fig. 1. The NFV-MANO high-level framework (adapted from [4]).

Removing the dependency between the network function software and the hardware infrastructure is expected to bring a variety of advantages in how networks are operated and managed [5], [6]. Nonetheless, it also brings additional implications on the network management systems that need to be extended beyond traditional FCAPS (Fault, Configuration, Accounting, Performance, Security) management services in order to provide life-cycle management of a new set of entities such as the VNFs, network services (NSs), and the virtualized infrastructure [7]. In addition, the operators need to ensure that service lifecycle is adequately orchestrated and managed such that service needs and requirements are met. To this aim, ETSI has defined a specific NFV-Management and Orchestration (NFV-MANO) framework [4], in the remainder simply referred to as MANO. Fig. 1 presents the high-level architectural view of the MANO framework which consists of the following functional blocks:

NFV Orchestrator (NFVO): It is the primary responsible for the orchestration and management of the NFV infrastructure (NFVI) resources across multiple virtualized infrastructure managers (VIMs) and the lifecycle of the network services including operations like on-boarding, instantiating, scaling, or terminating network services. It also interacts with the operation and business support system (OSS/BSS), through which customers/operators perform service operations including instantiating, updating, or terminating a service.

VNF Manager (VNFM): It is the block in charge of the configuration and lifecycle management of one or more VNFs.

The VNFM receives from the NFVO management instructions for VNFs (e.g. deploy, configure, and terminate) and executes them through its interfaces with the VNFs. The NFVO and VNFM jointly work to ensure that the network services and their corresponding VNFs meet the service quality requirements (e.g. reliability, latency or throughput).

Virtualized Infrastructure Manager (VIM): It manages and orchestrates the physical resources, i.e., compute, storage, and networking, upon which the VNFs are executed.

In addition to the three main blocks, a set of catalogs represent the repositories of on-boarded NS, VNF packages and the relative instances. Moreover, another repository holds information regarding available/consumed NFVI resources, as abstracted by the VIM.

An important end-user expectation is the high-availability level that NFV-enabled services will deliver. This is because several of the envisioned NFV service use cases fall into the telecom domain in which carrier-grade quality of service is a strict requirement, i.e., 5-nines availability [8], [9]. Moreover, NFV is foreseen to be a main pillar of future 5-th generation (5G) networks where stringent delay and availability demands (5-nines or more, i.e., less than 5 minutes of yearly downtime) are expected [10]. However, ensuring high-availability levels can be an arduous challenge that network operators need to cope with since service outages, induced by various component failures, are inevitable events. High availability is typically achieved by providing fault-tolerance capabilities through the allocation of redundant elements [11] over which the system switches upon the failure of primary components. To this end, a robust management and orchestration system featuring resiliency facets is mandatory for conducting correct and timely counter-actions to such events [12], [13]. Moreover, failures of the MANO itself could jeopardize the overall functionality of the network and potentially impact the service delivery by causing severe outages, which sometimes may be hard to deal with [14], [15]. It is thus of an utmost importance to ensure that a logically-centralized management and orchestration system is highly dependable and able to ensure *service continuity* [8]. To highlight the importance of a dependable MANO system, ETSI has published guidelines and requirements regarding the MANO resiliency capabilities [16].

Cloud-native application engineering is a consolidated approach in designing, building, and running applications that can fully exploit cloud computing benefits. An important pattern of cloud-native applications is that they are composed of microservices where each of these small services can operate independently of each other, provide a specific service, and communicate through well-defined mechanisms [17]. Moreover, cloud-native applications are packaged as a set of lightweight containers (e.g., Docker [18] or LXC [19]) aiming at providing context isolation, highly accessible, scalable and portable virtual environments. This way, service provisioning becomes more flexible, agile, and reliable [20]. Driven by such benefits, there is an increasing trend in adopting cloud-native design patterns also for virtualized network functions through deploying and running networking code as containerized software [21]–[23]. This trend has been embraced also by some of the most prominent open-source MANO projects which leverage a microservice architecture in deploying and operating MANO components through lightweight containers [24]–[26].

In this paper, we take a model-driven approach for predicting the availability of container-based MANO implementations and evaluating the impact that variations of critical failure and repair parameters have on the overall system availability. We adopt Stochastic Activity Networks (SANs) modeling formalism and perform a quantitative assessment of various deployment configurations enriched with fault-tolerance on both software and hosting infrastructure. An extensive sensitivity analysis allows us to localize bottleneck parameters for each of the deployment setups. The main contributions of this article introduce:

(i). Modeling abstractions for containerized MANO implementations, integrated with software rejuvenation and deployed in different redundant configurations, which are inspired by practices adopting cloud-native designs.
(ii). An experimental campaign on a containerized MANO platform aiming at retrieving realistic system recovery parameters.
(iii). A characterization of failure dynamics and an extensive sensitivity analysis targeting dependability metrics for both centralized and distributed MANO deployments.
(iv). Computational results that characterize failure dynamics, and sensitivity analysis that identifies critical parameters and rejuvenation policies for maximizing the steady-state availability (SSA).

The remainder is organized as follows. Section II presents the related work and highlights the key novelties. Section III presents the case study MANO architecture and the mapping of the components to the ETSI framework. The different deployment configurations that considered in this study are illustrated in Section IV. Section V introduces the software aging phenomenon and the mechanisms to cope with its related effects. The availability models resembling the different configurations are presented in Section VI. In Section VIII, we show the results of the analysis and conclude the paper by highlighting the most important insights in Section IX.

II. Related Work

NFV dependability is an important challenge and a significant research effort has been put on addressing this challenge. ETSI has promulgated various NFV specifications in regard to requirements, capabilities, and models for assessing reliability, availability, and service continuity [8], [16], [27], [28].

Most of the model-based studies evaluating NFV availability focus on network service availability modeling and quantification without considering the potential impact that the MANO may have on the end-to-end service availability. These studies either focus on specific NFV use cases such as virtualization of the evolved packet core (EPC) system [29] and the virtualization of the IP multimedia subsystem (IMS) [30], or model and analyze generic network services provided through NFV-enabled infrastructures [31], [32], without regarding the effect that a faulty MANO may have on the overall service availability. However, as emphasized by ETSI, the MANO plays a crucial role in fault management [16] and it may have a huge impact on the NFV-enabled network service performance [14], [15]. As a result, a study of its failure dynamics and availability analysis can be an important contribution for predicting and identifying MANO availability bottlenecks.

In [29], the authors present an availability model of a virtualized EPC by using stochastic activity networks. The study assesses the system availability through discrete-event simulation and identifies the most relevant criteria to account for by service providers in order to meet a certain availability level. The proposed model includes also the MANO system but no analysis is performed.

A two-level hierarchical availability model of a network service in NFV architectures has been proposed in [31]. By aggregating non-state space (Reliability Block diagrams) and state-space models (Stochastic Reward Nets), the authors quantify the SSA and perform a sensitivity analysis to determine the most critical parameters influencing the network service availability. Similarly, in [32], they extend such analysis by including the VIM functionality, as the entity responsible for the management of the physical infrastructure resources, into the reliability block diagram (RBD). Their main findings indicate that a relatively small increment of hypervisor or VNF software failure intensity has a marginal effect on the service availability. In addition, they identify the most appropriate redundancy configuration in terms of additional replicas for providing fine-nines availability. The same authors model and assess the availability of an NFV-oriented IP multimedia subsystem (IMS) [30]. Exploiting the same modeling techniques, they assess the availability of a containerized IMS and perform a sensitivity analysis on failure and repair rate of some of the IMS components. In addition, they identify the best k-out-of-n redundancy configuration for each IMS element such that a five-nine availability is reached.

In [33], the authors propose a hierarchical availability model of an NFV service by adopting stochastic activity networks. Each VNF, composing the network service, is considered as a load-sharing cluster and specific separate models abstracting different redundancy mechanisms, called Availability Modes, are constructed. The study performs a sensitivity analysis on various critical parameters and also investigates the impact that a faulty orchestrator has on the service availability. Differently, in this paper we focus on the MANO system rather than the NFV-service and propose availability models derived from current microservice based implementations. Moreover, our study provides insights on the most critical parameters specifically affecting the MANO availability for different deployment options and under software proactive maintenance.

Even though different from a model-based investigation, the authors of [34] propose centralized and distributed mechanisms for providing a reliable and fault-tolerant microservice-based MANO. The mechanisms exploit load balancing and state sharing and include some tunnable parameters which can help an operator optimise the trade-offs between reliability and the associated costs in terms of resource usage. The proposed setup allows the definition of a cost function which can help the operator determine the best configuration among the centralized and distributed MANO deployments.

One of the first studies to carry out an availability assessment of containerized systems is [35]. The authors propose availability models for different configurations and compare various container deployments. Through both analytic and simulation computational results they investigate the k-out-of-N redundancy configuration and evaluate the availability sensitivity to different failure parameters. In [36], the same authors present the development of a software tool called ContAv which can perform the evaluation of containerized systems' availability. Through the use of both non-state and state-space models, designed by the authors, the tool assesses the system availability for different configurations and allows a system architect to easily parametrize and perform sensitivity analysis. However, both works assume that container restarts are sufficient for recovering the containerized application. This can be an oversimplified assumption since the application source code, built in the container image, can also be subject to failures which require a software fix or patch [37]–[39]. Moreover, the work disregards the hardware infrastructure which can also be a dependability bottleneck despite the container instances are provided with instance redundancy. The models presented in our work relax these assumptions. In addition, we investigate also the impact that both aging and non-aging related bugs have on the system availability, where software rejuvenation is considered as a countermeasure.

Built on our previous attempt to characterize failure and recovery behavior of the MANO system [40], the present work extends the investigation in several aspects. One is more truthful modeling abstractions for MANO implementations. Another is a model for distributed MANO deployments which encompasses redundancy on both software and hosting infrastructure. In addition, a component-wise MANO model is introduced. In all these, the impact of software proactive maintenance, in the form of software rejuvenation, is particularly factored in. Moreover, we exploit fault-injection techniques and perform experimental trials on a realistic testbed based on which some key model parameters are retrieved for use in numerical analysis.

III. Case Study

There are currently several open-source MANO framework implementations, such as OSM [24], SONATA [26], and ONAP [41]. To restrain the nonconforming development of MANO architectures with incompatible APIs, ETSI has provided several guidelines of the different MANO architectural options [6], [42], which are currently widely accepted within the sector. Despite the various options, an ETSI-compliant architecture should adhere to the streamlined specifications and include the main functional blocks, which should provide an end-to-end network service management and orchestration.

In this paper we extrapolate the deployment options of OSM, a well-established architecture supported by ETSI and led by a large community of network operators and research institutions [24]. OSM claims to be closely aligned with ETSI NFV information models and consists in a production-quality and VIM-independent software stack. Eight releases have been distributed up to now and Release 8 is currently the latest release. It includes different installation methods

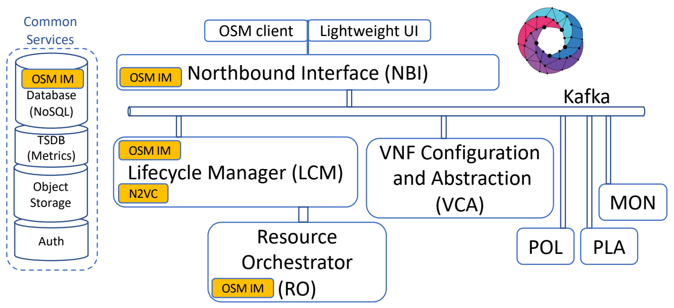


Fig. 2. OSM architectural view (adapted from [24]).

where the MANO components can be deployed as *dockerized* instances [18] into a hypervisor-based virtualized environment, a public hosting infrastructure, or directly into a proprietary commodity hardware. The latter represents a common way of deploying and running the OSM stack.

Fig. 2 illustrates the architectural view of OSM with the specific names of the stack components. The LCM module stands for Lifecycle Manager and plays the role of the NFVO in the ETSI MANO framework. The VCA assumes the role of the VNFM and exploits a Juju controller [43], deployed in a Linux Container (LXC) [19], for performing the VNFs configuration and management. The VIM, despite being formally part of the MANO framework, is typically bundled with the NFVI and thus is not present in the OSM stack. However, the interaction with the VIM is realized through a specific driver called resource orchestrator (RO). Note that this is also common for other MANO implementations, see for example OpenBaton [25] and Tacker [44]. A set of additional integrated components enable VNF placement, policy, fault and performance management. Specifically, the PLA component explores an optimization engine which defines the placement of VNFs into the available NFVI infrastructure, e.g., subject to resource constraints, cost, and utilization. The MON module performs monitoring by collecting VNF metrics from the VIM and VCA, storing them in a time-series database (TSDB), and reporting alarms related to these metrics. Policy management is accomplished by the POL component and regards tasks such as configuring auto-scaling groups for VNFs, listening for MON alarms, and reporting scaling/alarm messages to LCM when scaling/alarm conditions are met. In addition, there is also a set of common services such as data stores, authentication, and monitoring tools which are used by other components for accomplishing their tasks. For example, Prometheus [45] realizes the TSDB which is used to scrap and store time-series data related to VNF metrics collected by the MON module. Finally, the communication among the different components is executed through a unified distributed Apache Kafka message bus for asynchronous communication [46]. Apache Kafka is a fault-tolerant message queuing system that uses a publish-subscribe model for streaming messages like a data pipeline.

Typical operations that a standard-compliant MANO is expected to perform fall into five major categories [6]: i) VNF package-related operations such as on-boarding, enabling, disabling, updating, querying, and deleting VNF packages; ii) VNF-related operations such as feasibility check, instantiation, scaling (both expansion and contraction), terminating, and fault management; iii) NS descriptor (NSD) operations such as on-boarding, disabling, enabling, updating, querying, and deleting NSDs; iv) NS-related operations such as instantiation, scaling (scale-in and scale-out), updating, and terminating NSs; and v) VNF forwarding graph (FG), i.e., VNF chaining, lifecycle operations such as creating, updating, querying, and deleting VNF FGs.

Executing the aforementioned operations requires the cooperation of multiple functional blocks of the MANO framework. For example, the VNF scaling operations envision the coordination and exchange of control flows among the NFVO, the VNFM, and also the VIM [6]. This is also reflected in the OSM architecture since similar operations involve interaction of several components. As a mere example, the automated VNF scaling procedure relies on alarms, raised from VNF and VIM collected metrics, that trigger a scaling process for which also the MON, POL and TSDB components interact with the LCM, Juju and RO modules. Henceforth, from a dependability perspective, ensuring the complete functionality of the MANO requires that all components are able to provide their services. As a result, it is reasonable to assume the OSM software as a single entity since the failure of even a single component will prevent the system from providing its agreed function(s). This assumption is (to a certain extent) also validated in the experiments reported in Section VII and used in the analysis in Section VIII.

IV. Deployment Configurations

In this section we illustrate the different deployment cases which are the focus of this study.

A. Docker Swarm deployment

Docker is a widely used container technology and an application running on Docker is constituted by a container manager (also called engine or daemon), which manages images, volumes, networks, and container instances. A container instance is build from a container image which is typically stored in an image repository. It is common that for a given image, several container instances are spawned, forming a cluster, for purposes like load balancing, high-availability or scalability.

The OSM Docker swarm installation deploys 14 docker containers running in *swarm mode* with each component having one single replica. Docker *swarm mode* is a native feature of Docker for managing and orchestrating a cluster of Docker engines forming a so called *swarm*. It entails several cluster management characteristics such as: i) decentralized configuration of cluster nodes at runtime, ii) automatic scaling, iii) automatic cluster state reconciliation, and iv) integrated load balancing. A swarm is a cluster of Docker nodes which can act as managers, who manage the swarm membership and delegate tasks, and workers which run swarm services.

A Docker node can be a manager, worker, or both. A service is the definition of the tasks that shall be executed by the swarm through either standalone managers or worker nodes. When defining a service, the optimal state of it is defined by specifying features like number of replicas, network and

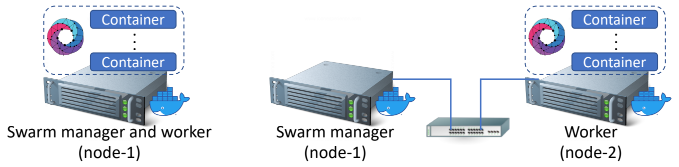


Fig. 3. Illustration of *Manager* (left) and *Manager-Worker* (right) deployment configurations and experimental testbed.

storage resources attached to it, and the ports the service exposes etc. It is the responsibility of the Docker manager to maintain the swarm state in case one of the worker nodes becomes unavailable by re-scheduling its tasks to other nodes.

A *swarm* may consist in only one node, which by default will simultaneously act as a manger and worker, but it cannot be only a worker without a manager. We refer to this setup as the *Manager* configuration. To be noted that this kind of deployment does not provide sufficient protection in terms of faulty physical host and supporting software like the operating system. Therefore, though not specifically recommended by the OSM community, we consider the case where an additional node joins the swarm for acting as a manager node and the service workload is only processed in the worker node. This is also a Docker recommendation in case a limited number of physical hosts is available [47]. In this case, the swarm cluster is composed of worker and manager nodes and we refer to it as *Manager-Worker* configuration. Fig. 3 depicts the key differences between the two deployment options.

One of the key features of a swam is the automatic cluster state reconciliation. This is an important feature in terms of fault management policies. In case one of the services of the cluster is down, the swarm state changes and the manager immediately respawns the failed container/containers on other available nodes (e.g., in the Manager node in a *Manager-Worker* setup) and the service stack becomes healthy again. Moreover, also in case events such as daemon, OS, and hardware failures are experienced on the worker node, all containers are respawned in the other node and the service is recovered.

B. Kubernetes deployment

Kubernetes, also known as K8s, is a container orchestration platform, alternative to Docker swam, created by Google and currently being managed by the Cloud Native Computing Foundation [48]. It was created with orchestration in mind and is supported by a much greater community compared to Docker swarm. In Release 8, OSM has evolved into supporting Kubernetes both as the infrastructure to run OSM as well as the infrastructure to deploy Kubernetes-based network functions.

Kubernetes is specifically designed for managing clusters of containerized applications. A K8s cluster consists of a set of worker machines, called nodes, and a container orchestration layer, called control plane. A worker node hosts the pods, which are the set of running applications executing the workload, and the control plane manages the worker nodes and the pods running in them. The control plane includes four components; the frontend K8s API server `kube-apiserver`,

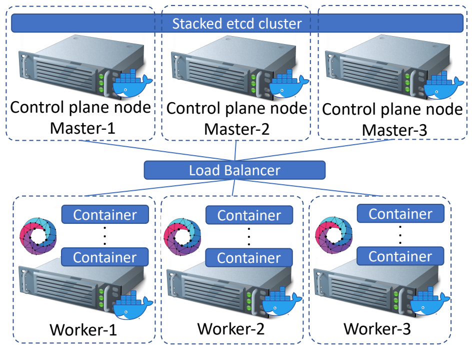


Fig. 4. Illustration of a highly available *Multi-master* cluster deployment.

the key-value data store `etcd`, `kube-controller-manager` process(es), and the task `scheduler`. Particularly important is the `etdc` system which is a strongly consistent and distributed key-value store for reliably storing data in a distributed system. It uses Raft consensus algorithm [49] for leader election and for ensuring that cluster internal state is consistently replicated among the members. For an N members cluster, the quorum, i.e., majority, is lost when more than $(N-1)/2$ members fail. For more details on how the Raft protocol operates, the reader may refer to [49].

A recent OSM feature is the ability to deploy OSM in a K8s highly-available (HA) cluster. In this deployment option, the OSM pods, i.e., OSM software stack components, are replicated into three distinct virtual machines running in the same physical hosts. In addition, also the control plane, called Master, is deployed in a separate machine and runs in the same host. This configuration aims at providing fault tolerance by actively running three OSM pods in a load-sharing configuration. In case any of the pods fails, the master will reschedule incoming requests on the remaining ones. However, fault tolerance is only on the OSM software level since the physical host is a single point of failure. Moreover, the failure of the the single Master would destabilize the cluster state and it would prevent the system from accepting and processing incoming requests although the pods would still be up and running.

To overcome this limitation, and driven by Kubernetes recommendations for deploying highly available clusters [50], we consider another topology, called *Multi-master cluster*, where worker and master nodes are distributed in multiple physical hosting nodes. The cluster is composed of three OSM pods which are deployed in separate physical hosts and there are also three Masters, forming the cluster control plane, with each of them also running in a separate physical node. Fig. 4 illustrates this K8s-inspired cluster topology. Each of the three Masters, hosts an `etcd` member and they together form an `etcd` cluster that enables maintaining a strongly consistent internal state and ensures that the lost of one of the members, i.e., Masters, can be tolerated. Note that only the Masters participate in the `etcd` cluster. This way, the failure of one single Master would not compromise the quorum and the cluster would still be able to elect a leader for managing the overall cluster.

V. Software Aging and Rejuvenation

Past studies of software engineering classify software faults into two main categories, Bohrbugs and Mandelbugs [37]. Bohrbugs, otherwise called deterministic, are software faults that typically can be easily reproduced since they tend to manifest themselves consistently under the same conditions. They often may lead to a software crash or process hanging and the bugs need to be identified and resolved. It is possible that accurate test and validation efforts can identify and correct this kind of bugs. Mandelbugs are bugs whose activation and error propagation are more complex in nature. They are difficult to isolate and as a result, they are hard to reproduce. Their manifestation is transient in nature and are usually caused by timing and synchronization issues resulting in race conditions. A retry operation or software restart may often resolve the issue [51].

Software aging is a well-known phenomenon associated with software systems [52]. The general characteristic of software aging is the fact that as the software execution time period increases, the associated failure intensity also increases. A successive activation of relative aging-related software faults causes software errors, which have not yet caused a software failure, to accumulate in the internal system state. It is due to this accumulation that aging-related errors may propagate to a system failure. This system state is also called the erroneous or failure probable state. It has been shown that all aging-related bugs are Mandelbugs [37], [52], hence further classifying Mandelbugs into two categories; aging-related and non-aging related Mandelbugs. Typical faults in IT software systems caused by aging effects include resource leakages, numerical errors, or data corruption accumulation.

The time to aging-related failure defines the time period from the moment of the software startup time to the observation of an aging-related failure. Its probability distribution is mostly influenced by the running lifetime period and the software workload quantity. The aging effect is not reversible without an external interventions and a proactive fault management method to deal with software aging is software rejuvenation. The rejuvenation aim is to clean up the internal system state and thus prevent the occurrence of more severe failures. Common methods of rejuvenation techniques consist of a system restart and/or reboot procedure [38]. Any rejuvenation will typically incur to some overhead, i.e., downtime due to safe restarts, but the goal is to prevent more severe crash failures that may be difficult to recover. As a result, an important problem is to optimize the rejuvenation schedule. Analytic-based models have been widely adopted to find the optimal tradeoff for a variety of software systems including virtualized servers [53]–[55], service function chains [56], and software-defined controllers [39], [57]. Common to all these efforts is the adoption of Petri-net based formalisms and the characterization of aging dynamics with the objective of identifying the optimal rejuvenation schedule such that the system SSA is maximized.

In similar lines, the scope of this work is not limited to characterizing MANO software-dependability dynamics impacted by the aging phenomenon but also assesses non-aging related faults' impact on the SSA. Henceforth, on the software level, we consider both aging and non-aging related Mandelbugs, while assuming that correct testing and validation has removed the Bohrbugs prior to deployment.

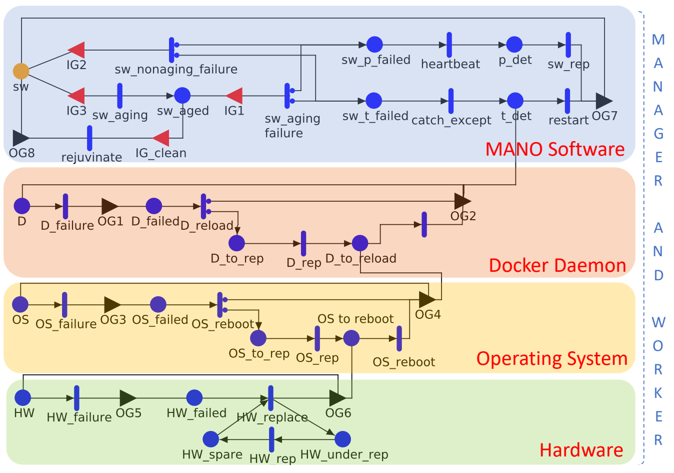
VI. Availability Models

A SAN is a modeling formalism with which detailed performance, dependability, or performability models can be implemented in a comprehensive manner [58]. SANs are stochastic extentions of Petri Nets consisting of four primitives: *places*, *activities*, *input gates*, and *output gates*. Places are graphically represented as circles and contain a certain number of tokens which represent the *marking* of the place. The marking of each place in the model represents the state of the system. Activities are actions that take a certain amount of time to fire and move tokens from one place to another. They impact the system performance and can be *timed* (thick vertical lines) or *instantaneous* (thin vertical lines). A timed activity has a distribution function associated with its duration and can have distribution case probabilities used to model uncertainty associated with activity completion. The case probabilities are graphically represented as small circles on the right of the activities. Upon completion, an activity fires and enables token movements from places connected by incoming arcs to places connected by outgoing arcs. This way a system state update occurs and tokens are moved from one place to another by redefining the places' markings. Input and output gates define marking changes that occur when an activity completes. Different from output gates, the input gates are also able to control the enabling of activity completion, i.e., firing. All the models are constructed using the Möbius software tool [59].

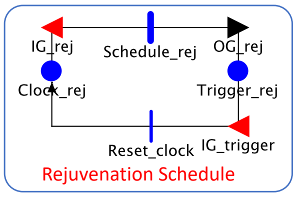
In the following, we illustrate the proposed abstraction models for the different MANO configurations.

A. Manager Configuration

Fig. 5 illustrates the SAN model of the *Manager* configuration. It abstracts the deployment of the MANO containerized software into one physical node, which acts as both manager and worker for the service tasks. Note that in the figure, we have treated the software deployment of both worker and manager together for illustration simplicity. Making the "manager" part more explicitly can be done similarly as in Fig. 6 for the *Manger-Worker* configuration. The model includes the MANO software (i.e., all MANO components), Docker daemon, OS, and hardware layers, and a similarly structured model may also apply to other containerized system. The places *D*, *OS*, and *HW* are initialized with 1 token each, indicating working Docker daemon, OS, and hardware components, respectively. The place *sw* is an extended place and allows the representation of structures or arrays. Specifically, we consider the tokens in *sw* to be a structure containing two fields, one representing the `operational units`, initialized with one token, and the other one representing the potential number of software `aging-related faults`, initialized with *N* tokens. Similarly to previous works (see [29]–[32]), it is assumed that all the timed activities follow a negative exponential distribution unless otherwise specified.



(a) SAN availability model of the MANO.



(b) Rejuvenation schedule model.

Fig. 5. SAN availability model of the *Manager* MANO configuration with software rejuvenation.

In [40] and some other studies (see for example [39], [57], [60]), software aging is modeled with a "one-shot" representation where a token is fired, following a certain distribution, from an up place to an error-prone place and the same token can be subject to a consequent firing due to a software aging-related failure. Nevertheless, this representation fails to capture the very essence of software aging, which is the continuous accumulation of software aging errors and the consequent increase of the failure rate. In this paper, we adjust this drawback by representing a more realistic aging behavior. Specifically, aging is represented through a timed activity *sw_aging*, with rate $\lambda_{\text{sw}_{\text{ag}}}$. The firing of *sw_aging* is enabled by the input gate *IG3*, which verifies that the system is operational, i.e., there is one token in the *sw* field `operational units`, and there is at least one token in the field software `aging-related faults`. For every *sw_aging* firing, there is a token removal from the N tokens, present in `aging-related faults`, and placed in *sw_aged*, which in turn represents the error-prone state. This way, the model allows the accumulation of aging errors in *sw_aged* and the *sw_aging_failure*, which represents the aging failure event, is directly proportional to the number of accumulated tokens in *sw_aged*. This way, the more accumulated aging errors, the higher is the failure intensity due to aging.

For the non aging-related Mandelbugs, the timed activity *sw_nonaging_failure* represents the non-aging related software failure event with rate $\lambda_{\text{sw-fail}_{\text{nag}}}$. When *sw_nonaging_failure* fires, the token representing the operational unit is removed from the place *sw* indicating that a MANO software failure has been experienced and the system is in a failed state.

For both software failure events, we differentiate between two types of failures based on their recovery process. We make use of case probabilities associated to the timed activities where C_{nag} defines the probability that a non-aging related failure event is recovered with a software restart and with probability $1 - C_{\text{nag}}$, the failure recovery requires a manual intervention for executing a software repair. Similarly, C_{ag} defines the probability that an aging related failure is recovered with a software restart and with $1 - C_{\text{ag}}$ with a software repair.

Once a software failure is experienced, a token is placed in either *sw_p_failed* or *sw_t_failed*, which define the recovery process that the software will undergo. *heartbeat* and *catch-exception* symbolize the detection of failures and are defined with deterministic times μ_h and μ_c. *sw_rep* and *restart* represent the repair (including any eventual reboot or upgrade of software) and restart events of the software with rate $\mu_{\text{sw}_{\text{rep}}}$ and $\mu_{\text{sw}_{\text{res}}}$, respectively. On the docker engine level, i.e., daemon, *D_failure* and *D_restart* model the failure and recovery events of the daemon with rates λ_{D} and $\mu_{\text{D}_{\text{r}}}$, respectively. The recovery entails a daemon restart where with probability C_{D} a daemon restart recovers the failure and with $1 - C_{\text{D}}$ a hard repair is needed. The latter is defined through the activity *D_rep* with rate $\mu_{\text{D}_{\text{rep}}}$. Once the daemon is repaired, an additional restart is performed to fully recover it. Similarly to the daemon, the operating system level is modeled with the same dynamics having specific failure and repair parameters which we introduce in Section VIII. On the hardware level, *HW_failure* and *HW_replace* represent the failure and recovery with rates λ_{HW} and $\mu_{\text{HW}_{\text{rep}}}$, respectively. The place *HW_spare* indicates the spare hardware equipment used to replace the failed hardware and is initialized with 1 token.

A novel contribution compared to our earlier work [40] is the adoption of software rejuvenation, as a proactive software maintenance mechanism. We apply a time-based rejuvenation where in specific time intervals, called rejuvenation intervals, the system undergoes a graceful software restart. To model this mechanism, we introduce a model (Fig. 10(a)) that defines the rejuvenation scheduling, and an additional timed activity *rejuvenate* models the time it takes the system to restart. More specifically, the place *Clock_rej* holds one token and the deterministic time activity *Schedule_rej*, which defines the rejuvenation interval, upon firing moves the token from *Clock_rej* to *Trigger_rej*, where the latter represents the state that the rejuvenation can be triggered. This movement is enabled by the *IG_rej* port which verifies that the system is operational, there is at least one token in *sw_aged*, and there is one token in *Trigger_rej*. If these conditions are satisfied, the rejuvenation is performed and *rejuvenate* fires a token. At the same time, *IG_clean* removes all the accumulated tokens in *sw_aged* by setting them to zero and sets the `operational units` field in *sw* to zero, indicating that the system is undergoing a downtime due to rejuvenation. The *Schedule_rej* and *rejuvenate* activities are defined with deterministic times μ_{Sched} and μ_{rej}, respectively. Once the rejuvenation is completed, the token is moved from *Trigger_rej* and placed into *Clock_rej* by the firing of the instantaneous activity *Reset_clock*, and the output gate *OG8* resets the *sw* fields `operational units` and `aging-related faults` equal to one and N tokens, respectively. The output gate *OG7*

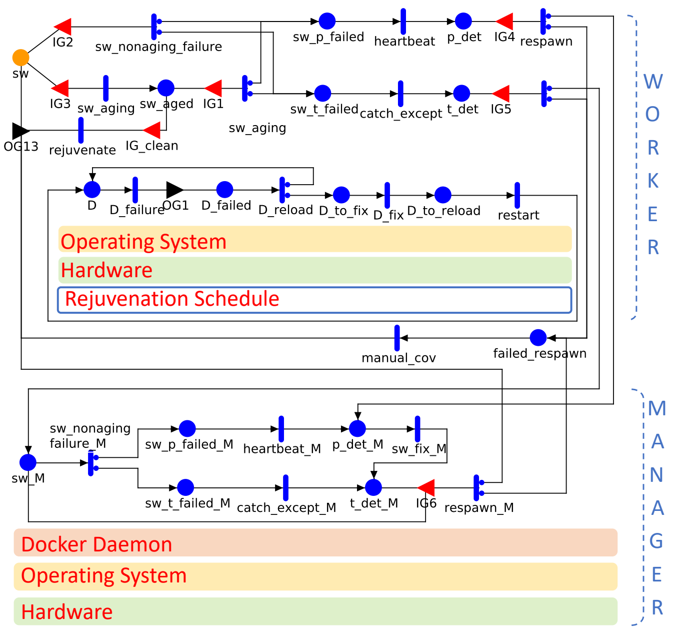


Fig. 6. SAN availability model of the MANO deployed in a *Manager-Worker* configuration.

operates similarly to *OG*8 except that in this case the system has gone through a software recovery procedure. Note that rejuvenation can be performed only when it is scheduled to happen and the system is operational.

Finally, the following output gates define the token marking movements among lower-level places: *OG1/OG3/OG5* manage the failure events of the daemon, OS, and hardware levels, respectively. When their related timed activities fire, connected to their incoming arcs, the output gate places one token in the respective failed position and sets to zero the upper-level places. This is because a failure of the physical hardware will cause a failure of the OS which in turn impacts the operational state of the daemon and MANO software as well; *OG2/OG4/OG6* places 1 token in their relative working place, i.e., *D/OS/HW*, and the relative upper-level places to which they are connected by outgoing arcs. For example, a recovery from a daemon failure brings the daemon in the up state but requires a restart of the MANO software for a fully working system. The system is fully operational when the `operational units` field of *sw* place holds one token.

B. Manager-Worker Configuration

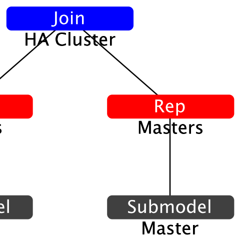
The *Manager-Worker* configuration consists of two separate nodes forming a cluster and the OSM stack is deployed on the worker node, with the latter being responsible for workload processing. Fig. 6 depicts the *Manager-Worker* SAN model. To distinguish the models of the two entities, we add a suffix *_M* for all the places and activities regarding the manager part. The system is fully working if there is a token in either of the *sw*, *sw_aged*, or *sw_M* places.

On the worker node, the MANO software component is similar to the *Manager* configuration except for the recovery phase where once a failure is detected, the containers running the software are respawned, through the timed activity *respawn*, in the manager node. We distinguish two cases: when a software repair is needed, the token is moved from *p_det* of the worker node to *p_det_M* of the manager. In the other case, the token is moved from *t_det* to *sw_M* indicating that a respawn, i.e., container restart, is sufficient to recover the system. However, for both cases, we consider the eventuality of a respawn process that fails. To this end, we consider two case probabilities associated with the timed activities. With probability C_{respawn}, the container respawn is successful and $1 - C_{\text{respawn}}$ it fails. In the latter, there is a need for a manual coverage, represented by *manual_cov*, and the token is placed back in place *sw*. In order for the respawn to instantiate, the hosting manager node needs to be operational and this is controlled by the enabling gates *IG1/IG2* which enable the respawn only if the daemon, OS and hardware of the manager are working, i.e., their respective places *D_M*, *OS_M*, and *HW_M* contain each 1 token. In addition, differently to the *Manager* setup, once the daemon fails, there is just the recovery of the daemon since the MANO software is immediately respawned in the manager node. The rest of the model is similar to the *Manager* configuration and due to space limitations we use colored bars with component names to indicate the relative parts of the model and omit illustrating.

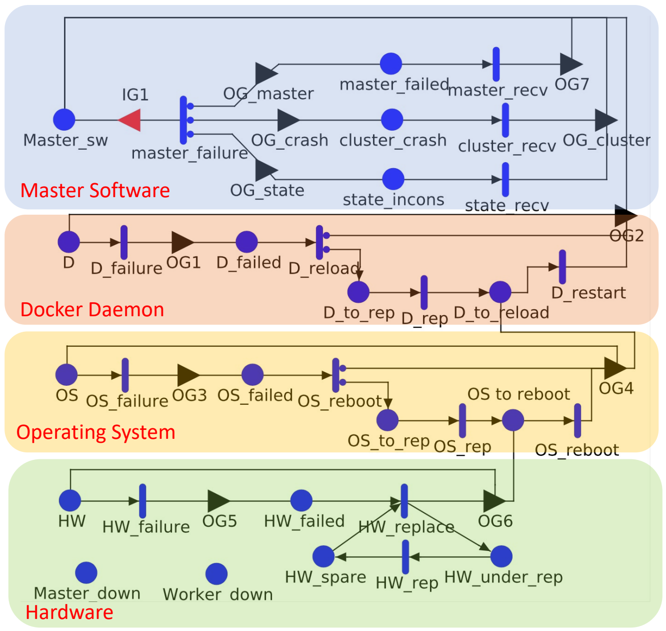
On the manager node, once a token is deposited in *sw_M*, the system is again operational. While the software is running in this node, we assume that it is subject to only non-aging software related failures. This is because *swarm mode* best practices suggest that the worker node should be the dedicated node for handling task requests in a 'normal' condition. Therefore, we limit the hosting of the MANO software to the manager node only for the period the worker node is failed. To this end, the input gate *IG6* enables a respawn of the software containers from the manager node to the worker node once the worker node is up and running again and ready to accommodate the containers. As a result, the manager node will host the containers for a relatively short time compared to the software aging time, hence making the assumption of only non-aging failure events on the manager node a reasonable assumption. The rest of the manager components, i.e., daemon, OS, and hardware are similar to the *Manager* configuration which for lack of space have been represented through colored bars, hence we omit further illustrating.

C. Multi-master Cluster Configuration

For abstracting the Multi-master cluster system, we exploit a Rep/Join model composition formalism which is integrated in Möbius. The formalism exploits system symmetries and generates lumped state spaces which are smaller compared to systems that do not exploit symmetries. This is particularly useful for large systems whose model nets generate complex stochastic processes [61]. The formalism enables the composition of a model in the form of a tree, where each leaf node represents a system submodel and each non-leaf node can be a Join or Replicate node. A Join node is a state-sharing node used to compose two or more submodels, whereas a Replicate node is used to compose a model consisting of a number of identical submodel replicas and can also enable



(a) Composed model.



(b) Master SAN model.

Fig. 7. SAN availability model of the MANO deployed in the *Multi-master* cluster with software rejuvenation.

state-sharing among its replicated submodels. The replicated submodels behave independently of each other and the root node represents the complete cluster model.

The *Multi-master* cluster we consider is not part of a deployment option or an enhancement feature of OSM and is primarily driven by Kubernetes recommendations for the deployment of 'truly' highly available clusters [50]. For the scope of our investigation, we make some reasonable assumptions that limit the system complexity, yet do not impact system performance, as they can be deployment options that an operator can arbitrarily choose. First, we assume that the cluster components fail independently. This can be a reasonable assumption in case components are geographically distributed; therefore, minimizing the likelihood that events can simultaneously affect two or more nodes. In addition, we assume that the load balancer is failure free and uniformly distributes the workload among nodes (refer to Fig. 4). Moreover, we also consider that the OSM pods are not deployed in virtual machines but directly on standard hardware running an operating system. We also assume that each worker node of the cluster runs a Docker runtime engine, i.e. daemon.

The cluster is modeled through the Rep/Join formalism by replicating three times both the Master, i.e., control plane, and the Worker submodels, as illustrated in Fig. 7(a).

The Worker submodel is similar to the *Manager* configuration model except for the presence of two shared places called *Worker_down* and *Master_down*. These two places are also present in the Master submodel and are used to keep track of the availability of Workers and Masters for the overall composed model, i.e., the *Multi-master* cluster. Every time a Worker or Master fails, a token is placed in the respective place and removed when they are recovered.

The Master submodel, illustrated in Fig. 7(b), is similar to the previous configurations on the hardware, OS, and Docker daemon levels. On the software level, we consider failure events that can affect either singularly the Master nodes or the overall cluster. Several studies have shown that distributed applications experience a variety of issues due to their distributed implementation. Some of the most typical issues that can cause cluster-wide failures concern state inconsistencies, leader election, defective fault management, or scalability issues [39], [62]–[64]. We account for these failure modes by assuming that each of the Master replicas may experience software failures (e.g., failures of the API server, scheduler, or etcd members) causing a single replica failure, cluster-wide crash, or cluster state inconsistencies. We use state distributions to characterize these events with probability C_{master}, C_{crash}, and $1-C_{\text{master}}-C_{\text{crash}}$, respectively. The Master software fails with rate λ_{Master} (transition *master_failure*), and this event is enabled through the input gate *IG1* only if less than three Masters are down, i.e. less than three tokens in the shared place *Master_down*. In case a cluster-wide crash or state inconsistency is observed, the respective output gates *OG_crash* and *OG_state* place three tokens in *Master_down*. On the recovery of such failures, the gate *OG_cluster* removes three tokens from *Master_down* and places them in *Master_sw* to indicate that a cluster-wide failure has been recovered.

The overall *Multi-master* cluster is considered unavailable when three tokens are present in place *Worker_down* and more than one token is present in place *Master_down*. Note that also other failure/recovery events on the other components (daemon, OS, and hardware) for both submodels place/remove one token in *Master_down* or *Worker_down* depending on the submodel.

D. Component-wise MANO Model

The approach taken so far in this paper is to consider the MANO software as a single component on the software level. On the one hand, decomposing the MANO software model into specific components would allow characterizing the various components in a finer grain in regard to their failure/repair dynamics. This can be of particular interest for cases where some software components are developed, tested, and validated by 'external' developing teams which may follow different practices, as is the case of the Juju VCA component which is developed and maintained by Canonical rather than the OSM community. Nevertheless, abstracting realistic MANO solutions is still subject to the actual architecture since the various solutions significantly differ in terms of architecture and implementations [65]. To illustrate, modeling the OSM software would require abstracting 14 software components, and even more for ONAP because it comprises 20 functional modules [65]. As a result, it is hard to employ a generalized model which is capable of a fine grain modeling of realistic implementations.

On the other hand, at a high level, all solutions should adhere to the ETSI standards, where the three main functional blocks, i.e, NFVO, VNFM, and VIM, must be part of a compliant architecture. This requirement can be reflected in a functionality-wise generalized model. This modeling approach could be suitable in cases where failure/repair dynamics of individual components differ significantly, though the lack of detailed studies in this matter, and ultimately failure and repair parameters of individual components, may discourage the pursue of this modeling approach. In the rest of this section, such a component-wise modeling attempt is introduced. In the next section, we also introduce experimental trials to retrieve key parameters regarding recovery times of individual components which can be used for a preliminary investigation.

Fig. 8 depicts the adopted model of a high-level MANO with separate components. In particular, the model includes separate NFVO, VNFM, and VIM software elements, deployed in the same hosting node, and their relative rejuvenation policies. The same software layer model utilized in the *Manager* configuration is used to abstract each of the components and due to lack of space they are represented through colored bars instead of SAN primitives. The three components are assumed to fail independently and the failure of just one of them would lead to an unavailable MANO. This assumption is according to the expectations of an ETSI-standardized NFV-MANO system as all main functional blocks are expected to be fully operational in order to be able to orchestrate and manage NFV services [16]. Due to the lack of failure data regarding MANO solutions, let alone single software components, we assume that each of the components is characterized by similar failure times which together exemplify the total intensity of the MANO software adopted in the other models. Regarding the repair process, we retrieved individual recovery parameters through experimental trials on the OSM solution by injecting faults on the software level targeting individual components, i.e., LCM, Juju VCA, and RO. On the Docker daemon, OS, and hardware level, the same submodels utilized in the *Manager* configuration are also used here, and the failure of any of these levels demands a restart of each of the MANO components once the level is restored such that the system can be deemed operational.

The rejuvenation process is separate for each of the three components and is subject to the individual utilization and software aging rates. For example, an operator could reduce the rejuvenation frequency for less utilized components and vice-versa. However, for simplicity, and also due to lack of knowledge regarding individual failure characteristics, we assume that the same rejuvenation process governs the individual rejuvenation policies. This can also be beneficial since a fully synchronized rejuvenation process will lead to the minimum downtime overhead introduced by rejuvenation. In addition, the rejuvenation duration is equal to the highest amount of time required to restart the single components. The model is solved by feeding the individual recovery times of the LCM, Juju VCA, and RO components, while maintaining the total failure intensities. We compare the two approaches in terms of SSA and perform a sensitivity analysis on the impact that variations of the rejuvenation interval, software aging, and software-aging induced failures have on the SSA.

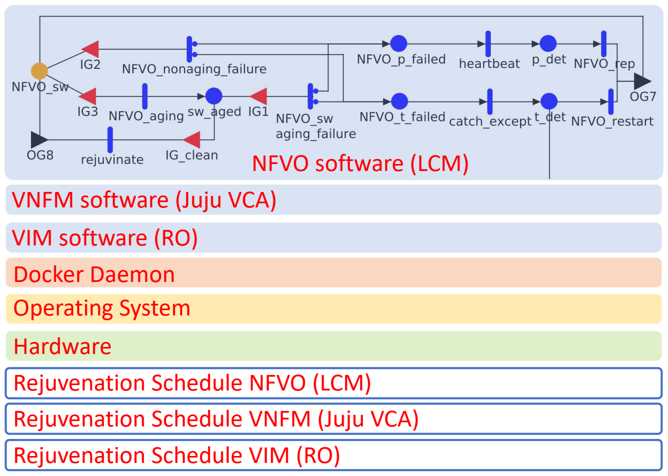


Fig. 8. SAN availability model of the MANO with separate software components.

VII. Experimental Testbed

Model-driven availability assessment relies on model parameters regarding failure and repair processes. However, the lack of failure and repair data is a common issue for novel technologies and projects. To partially tackle this issue, we performed an experimental campaign aiming at retrieving realistic recovery times of the system components by adopting fault-injection techniques. Our testbed consists of hardware and software technologies that are commonly used in cloud computing infrastructures in which the OSM software stack is deployed from scratch. Specifically, OSM Release 8 is deployed in *swarm mode* option, i.e., with Docker swarm orchestrator, into a Linux-based operating system (server version with kernel 5.15) with Docker engine (version 20.10.5) running on a 56-core Intel® Xeon® @ 1.70GHz machine with 128GB RAM, two 10-Gbps and two 1-Gbps Intel Ethernet NICs, and four 1-TB SATA hard drives. In this deployment option, the single machine will act as both manager and worker node, i.e., *Manager* deployment.

In order to perform measurements of the *Manager-Worker* deployment in case the worker node experiences failure on the host level, namely respawn times, we join to the OSM swarm deployment another host machine by using the standard `docker swarm join` command. The latter node is equipped with the same software and hardware technologies of the previous one and acts as a worker node. The host machines are connected to each other by their 10-Gbps NICs through a 5-Gbps Ethernet network switch and the OSM swarm is deployed in the worker node. Fig. 3 depicts the testbed adopted for the experimental campaign. This way, we emulate the two Docker swarm deployments and the testbeds are ready for fault injections on the different system components.

Starting with the *Manager* deployment, we inject the following fault types:

Software faults: responsible for software crashes and process hanging of the OSM software layer. Such faults can be varied in terms of manifestation nature including time and synchronization issues resulting in race conditions, resource leakage due to software aging errors, and error handling faults [37], [51]. Several of these software faults are also reported in the Bugzilla bug tracker platform utilized by the OSM community [66]. To emulate the occurrence of these faults, we forcefully terminate each of the OSM containers, and measure the time it takes for the stack to return in a running state. Precisely, we kill all containers of the stack and continuously (every second) interrogate each of the tasks, i.e., containers, until they reach a running state (the `.CurrentState` of the task). The interval between the time the fault is injected and the time the last task is running defines the overall time that will parametrize the mean time to perform an OSM software restart (i.e., the μ_{restart} activity on the model).

Docker engine faults: similarly to OSM containers, also the Docker engine can be subject to software faults. [67] reports faults affecting the Docker engine caused by software aging phenomenon. This component is particularly critical as a failure of the daemon causes the simultaneous failure of all running containers, networks, and mounted volumes. We mirror the fault on this layer by abruptly halting the container management process, i.e., `dockerd` process, and record the time it takes to restart, i.e., be running again. The measurements will define the rate of the *D_restart* activity.

Operating system faults: also the operating system is affected by software faults and several studies present recurring faults including OS exceptions, error codes, OS panics, or hangs [68]–[70]. Needless to say, the failure of the OS results in the termination of all the software layers running on top. To mimic this type of faults we force an immediate OS reboot without terminating any process or unmounting any file systems, i.e., hard reboot. The experiment executes the reboot command and records the time the command is issued. Upon system boot, we retrieve the time it takes for the kernel to reach the default runlevel (5 in the machines) and compute the time difference. The assessment will determine the mean time to perform an OS reboot, defined as μ_{OS_r} in the models.

Swarm node faults: these are faults that trigger a respawn of the containers in another node in case events such as daemon, OS, and hardware failures are experienced on the node that hosts the swarm services. We emulate this kind of faults by using standard docker commands that drain the availability of the node to host the containers and this triggers the automatic re-instantiation of the whole stack into another node. Specifically, we run `docker node update --availability drain <NODE-ID>` on the *Worker* node, which disables the *Worker* ability to host swarm tasks, and measure the time it takes for all containers to reach a running state in the *Manager* node.

The considered form of injection is focused on failure modes as effect of faults occurring in the components that can affect the system. Although from a terminology viewpoint, this form of injection in some cases is referred to as *error/failure injection*, it is also common to refer to this form as *fault injection* since failures of a component can be regarded as faults from the perspective of the system that incorporates the component [71].

We performed 50 controlled experiments for each fault type, resulting in 200 experiments in total. For each fault type, we develop ad-hoc shell scripts that inject the fault, trigger the recovery and measure the recovery time, wait for a reasonable amount of time such that the targeted system reaches a stable state, and re-run the fault injection. It is worth noting that we consider these kinds of faults as events that cause a soft failure of the targeted system for which a restart/reboot of the system is sufficient to recover it. In addition, we also performed 50 fault-injection trials individually on three software components; the LCM module, the Juju VCA, and the RO component. These individual mean recovery times are used in the assessment of the *Component-wise* model.

While running the experimental trials, we made several observations. At first, through an inspection of each of the containers, using `docker inspect` command, we observe that, while each of the containers is created within seconds from the fault injection time, the times for them to reach a running state significantly differ from each other. Some tasks reach a running state within a few seconds, e.g., the Database and the AUTH components, while others require a few tens of seconds, e.g., the RO, POL, and TSDB. Other components require even a few minutes to reach a running state, hence clearly showing a significant difference compared to recovery times reported in studies regarding containerized applications (i.e., recovery within hundreds of milliseconds) [35], [36].

Secondly, we observe a consistent behavior when inspecting the faults that cause a restart of the whole OSM stack, e.g., *Docker engine faults*. The LCM and the MON containers are always the last to reach a running state, with LCM reaching the desired state before MON. Although they are started multiple times, they fail to reach a running state until the rest of the components are running. Such observation is different when the single components of LCM, RO, and Juju are restarted individually. The times in these cases are smaller, refer to Table I, and consequently lead to an intuition that there should be some software dependencies among the components such that only when other containers are running, and consequently exposing services, others may reach a running state. However we are not able to identify the level of dependency for each of the running containers without a detailed knowledge of the software architectural design. This observation further supports the consideration that treating the OSM stack as a single entity may be more reasonable than treating its individual elements separately. Finally, during the *Swarm node* fault-injection measurements we observed that upon the node availability draining, all the containers were quickly respawned in the other node except Grafana and Prometheus. This behavior led to swarm instability, and hence we applied a workaround by quickly rolling back the node availability so that these two components can be restarted in the same node. This is likely due to some dependency among these components and the host node where they are initially launched. Clearly this does not represent the considered scenario, i.e., *Manager-Worker*, but we assume that their respawn times are similar, although respawned in the same node. We measured the respawn times similarly to the *Manager* case by adopting the workaround.

The mean recovery times described above, together with the relative standard deviation, are reported in Table I. We notice that some of the components such as the Juju, the RO and the Docker daemon have a rather fair stability in their mean time to recover since they present a limited spread of time values. As expected, the restart of the OSM software on both the same or another node, i.e, *Swarm node* faults, presents very similar values. This is because the services are managed by the swarm and spawning containers in another node, with the same processing capability, involves the same process, i.e., the docker engine spins up the same tasks using the already pulled container images.

VIII. Numerical Analysis

In this section, we present a numerical (evaluation) study. The proposed models are defined in the Möbius software tool [59] and they are solved using discrete-event simulation, integrated in the tool, with 99% confidence interval and 10^{-5} width of relative confidence interval. The SAN model parameters are in part retrieved from previous literature [39], [57], in part from experimental measurements, and the rest are estimated guesses based on our empirical experience. They are illustrated in Table I, and they represent the baseline parameters.

TABLE I
Availability model parameters
($^{\diamond}$from experiments, ‡from literature [39], [57]).

Intensity	Time	Description [Mean time to]
$\lambda^{-1}_{\mathrm{sw}_{\mathrm{ag}}} = 1$	week	MANO software aging‡
$\lambda^{-1}_{\mathrm{sw-fail}_{\mathrm{ag}}} = 3$	days	next MANO software failure after aging‡
$\lambda^{-1}_{\mathrm{sw-fail}_{\mathrm{nag}}} = 1$	month	next MANO non-aging software failure‡
$\mu^{-1}_{\mathrm{sw}_{\mathrm{rep}}} = 1$	hour	MANO software repair‡
$\mu^{-1}_{\mathrm{sw}_{\mathrm{res}}} = 185\ (\pm 15.6)$	seconds	MANO software restart (OSM stack)$^{\diamond}$
$\mu^{-1}_{\mathrm{NFVO}_{\mathrm{res}}} = 32\ (\pm 3.1)$	seconds	NFVO container restart (LCM)$^{\diamond}$
$\mu^{-1}_{\mathrm{VNFM}_{\mathrm{res}}} = 8.5\ (\pm 0.6)$	seconds	VNFM container restart (Juju VCA)$^{\diamond}$
$\mu^{-1}_{\mathrm{VIM}_{\mathrm{res}}} = 19.5\ (\pm 0.7)$	seconds	VIM driver container restart (RO)$^{\diamond}$
$\mu^{-1}_{h} = 10$	seconds	heartbeat*‡
$\mu^{-10}_{c} = 1$	millisecond	catch exception*‡
$\lambda^{-1}_{\mathrm{D}} = 4$	months	next daemon failure‡
$\mu^{-1}_{\mathrm{D}_{\mathrm{rep}}} = 1$	hour	daemon repair‡
$\mu^{-1}_{\mathrm{D}_{\mathrm{r}}} = 30\ (\pm 1.8)$	seconds	Docker daemon restart$^{\diamond}$
$\lambda^{-1}_{\mathrm{OS}} = 4$	months	next OS failure‡
$\mu^{-1}_{\mathrm{OS}_{\mathrm{rep}}} = 1$	hour	OS repair‡
$\mu^{-1}_{\mathrm{OS}_{\mathrm{r}}} = 249\ (\pm 21.4)$	seconds	OS reboot$^{\diamond}$
$\lambda^{-1}_{\mathrm{HW}} = 6$	months	next hardware failure‡
$\mu^{-1}_{\mathrm{HW}_{\mathrm{rep}}} = 24$	hours	hardware repair‡
$\mu^{-1}_{\mathrm{HW}_{\mathrm{replace}}} = 1$	hour	hardware replace‡
$\mu^{-1}_{\mathrm{rej}} = 3$	minutes	rejuvenation duration$^{\diamond}$
$C_{\mathrm{nag}} = 0.3$		prob. for non-aging transient failures‡
$C_{\mathrm{ag}} = 0.7$		prob. for aging transient failures‡
$C_{\mathrm{D}} = 0.9$		daemon restart coverage factor‡
$C_{\mathrm{OS}} = 0.9$		OS reboot coverage factor‡
$N_{\mathrm{spare}} = 1$		Number of spare hardware‡
$N = 60$		Number of potential software aging faults‡
$\mu^{-1}_{\mathrm{respawn}} = 189\ (\pm 21.6)$	seconds	respawn MANO software containers$^{\diamond}$
$\mu^{-1}_{\mathrm{cov}} = 1$	hour	manual coverage
$C_{\mathrm{respawn}} = 0.9$		respawn coverage factor‡
$\lambda^{-1}_{\mathrm{Master}} = 4$	months	next master failure
$\mu^{-1}_{\mathrm{master}} = 5$	minutes	recover master failure
$\mu^{-1}_{\mathrm{cluster}} = 15$	minutes	recover cluster crash
$\mu^{-1}_{\mathrm{incons}} = 3$	minutes	recover state inconsistencies‡
$C_{\mathrm{master}} = 0.55$		probability of Master software failure‡
$C_{\mathrm{crash}} = 0.05$		probability of cluster-wide crash‡

*Deterministic time

TABLE II
Steady-state availability of the different models without and with optimal rejuvenation policy.

	Manager	*Manager-Worker*	*Multi-master*	*Component-wise*
MANO w/o rej.	0.99723	0.99871	0.999782	0.99723
MANO opt. rej.	0.99762	0.999215	0.999915	0.99794
Downtime reduction	14%	39%	61%	25%

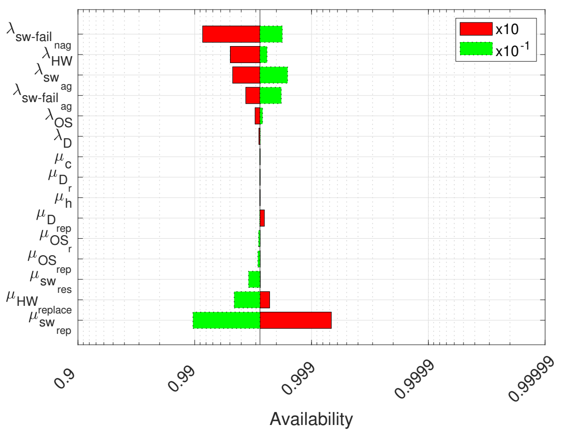
A. Sensitivity Analysis

Given the baseline parameters, the achieved MANO availability for every model is presented in Table II, together with the relative availability when an optimized rejuvenation policy is applied. We observe that for all deployments there is a meaningful improvement in terms of downtime reduction when an optimal rejuvenation policy is applied. The gain is more pronounced for the *Manager-Worker* and *Multi-master* case studies, achieving a 39% and 61% of downtime reduction relative to the system downtime without rejuvenation.

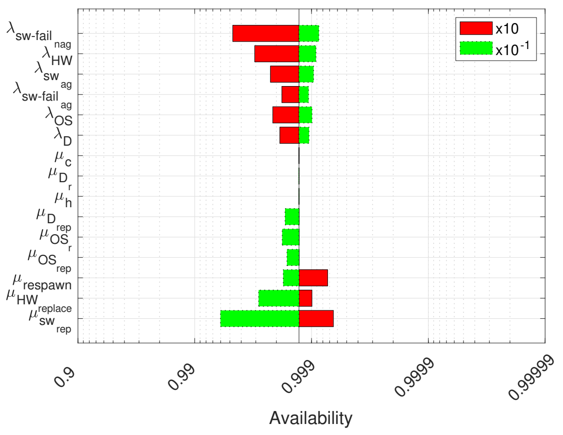
The sensitivity analysis is performed by varying failure and recovery parameters with one order of magnitude, i.e., $\times 10$ and $\times 10^{-1}$, from their baseline values, and retrieving the SSA in case no rejuvenation is employed. The sensitivity to these parameters, for all the case studies, separated into failure and recovery events, is presented in Fig. 9. We have adopted a modified logarithmic scale on the availability axis in order to obtain a better visualization of the high availability numbers.

For the *Manager* case, the most impactful failure parameters are software non-aging failure rate followed by hardware, software aging, and software aging failure rate. In particular, reducing the software non-aging related failure rate decreases the availability to 0.9913. Among these failure parameters, software aging rate brings the highest improvement on the SSA, reaching 0.9984. Concerning recovery parameters, software repair, followed by the hardware replace rate, has the highest impact on the system availability by reducing it from 0.99723 to almost 0.989. At the same time, the highest improvement, reaching 0.99932, is achieved for a software repair rate increase, i.e. lower software repair time.

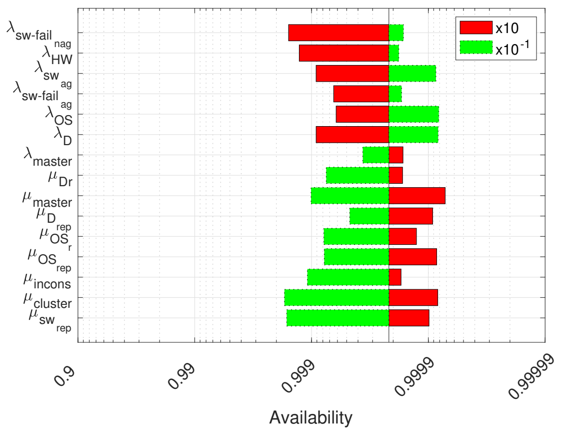
The same analysis for the *Manager-Worker* deployment reports a considerable reduction of the critical parameters. Besides the overall availability gain introduced by the fault-tolerance on the host level (refer to Section VI-B), the negative impacts of both software non-aging failure rate and software repair rate are markedly reduced compared to the *Manager* case. To illustrate, for the *Manager* case, a ten-fold increase of software non-aging related failure rate decreases the SSA from 0.99723 to 0.99143, which corresponds to an increase of 50.84 hours of yearly downtime (from 24.28 to 75.12). For the same parameter reduction, the *Manager-Worker* SSA is reduced from 0.99871 to 0.99526 corresponding to a 30.76 hours of additional yearly downtime. However, there is a more evident impact of parameters related to the physical host. We notice that an increase of either the OS or daemon failure rate has a more pronounced effect on availability reduction. This is because more frequent failures of the Manager OS or Docker daemon would prevent the Manager from hosting the MANO software in case a failure is observed on the Worker side.



(a) *Manager*



(b) *Manager-Worker*



(c) *Multi-master*

Fig. 9. Sensitivity analysis for the different MANO deployments without rejuvenation.

Moreover, a higher respawn rate (μ_{respawn}) can considerably improve the SSA up to 0.99937.

The *Multi-master* configuration increases further the system availability, reaching 0.999782. However, the same critical parameters identified in previous case studies continue to be critical. In particular, similarly to the *Manager-Worker* setup, deterioration of host level failure intensities has a non-negligible impact. This is because such failures influence more the availability of the Master nodes compared to the Worker nodes due to the fact that failure of more than one Master limits the overall cluster availability, as opposed to the Worker nodes where failure of all the three replicas is needed to cause a service outage. On the other hand, a ten-fold improvement, whether failure rate reduction or recovery rate increase, brings more substantial benefits in the SSA. For several failure and recovery parameters, SSA values exceed four nines availability, i.e., less than 52 minutes of yearly downtime. Observing individual Master parameters, we notice that the recovery times of events that can cause a cluster-wide failure such as μ_{cluster} and μ_{incons} greatly impact the system SSA. In particular, a ten-fold change of the time to recover cluster failures can affect the SSA significantly. This is because recovering a cluster crash requires a larger amount of time compared to the events that cause inconsistent states. On the other hand, a ten-fold reduction in μ_{incons} also causes a comparable reduction which can be explained by the higher frequency that such events occur. These findings confirm past model-based assessments of distributed control-plane implementations which report the impact of cluster-wide failures [39].

B. *Software Rejuvenation Impact*

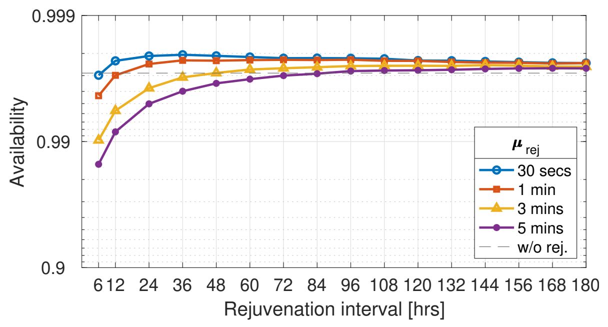
It is obvious that applying frequent rejuvenation does prevent the accumulation of aging errors, yet a frequent maintenance may lead to useless downtime caused by software restart. In order to fully profit from rejuvenation, an operator needs to find a balance between the deliberate downtime and the avoiding of more severe outages due to the error accumulation. Therefore, an operator should determine the optimal policy for scheduling the rejuvenation process.

The impact of different rejuvenation policies, i.e., μ_{Sched} and μ_{rej}, for all the cases under study are illustrated in Fig. 10 and Table III summarizes the optimal values. We use the same modified logarithmic scale to better illustrate the computed results. As expected, and common to all models, the results show that that enabling a shorter rejuvenation duration, brings significant benefits in availability. This is more evident when early rejuvenation schedules are applied.

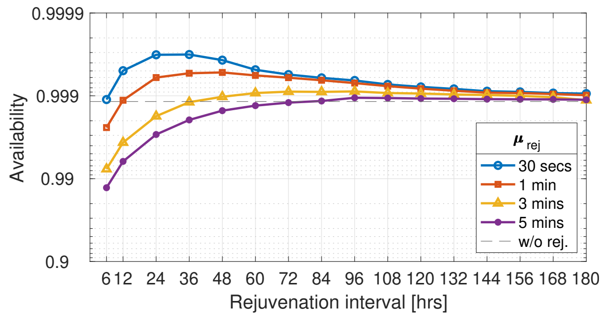
TABLE III
STEADY-STATE AVAILABILITY WITH OPTIMAL REJUVENATION POLICY WHEN VARYING REJUVENATION DURATION.

Rejuvenation duration	**Availability [Optimal rejuvenation interval in hrs]**			
	Manager	***Manager-Worker***	***Multi-master***	***Component-wise***
30 secs	0.99795 [36]	0.999678 [36]	0.9999362 [48]	0.99794 [48]
1 min	0.99785 [72]	0.999428 [48]	0.9999320 [60]	0.99772 [72]
3 mins	0.99762 [132]	0.999215 [96]	0.999915 [72]	0.99715 [180]
5 mins	0.99737 [168]	0.999144 [108]	0.999894 [84]	0.99705 [180]

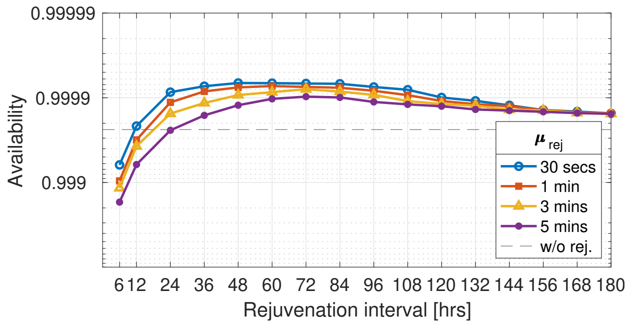
Concerning the *Manager* scenario for baseline parameters, i.e., a rejuvenation duration of 3 minutes, the maximum achievable availability is 0.99762 with an optimal rejuvenation interval of 132 hours. In case the software maintenance lasts longer, i.e., 5 minutes, the optimal rejuvenation trigger time is 168 hours, yet the availability gain is almost negligible compared to the non rejuvenated case. A similar trend is observed for the *Manager-Worker* implementation. The maximum availability that can be achieved with baseline parameters is 0.999215 for a maintenance interval of 96 hours, i.e., a safe software restart every four days.



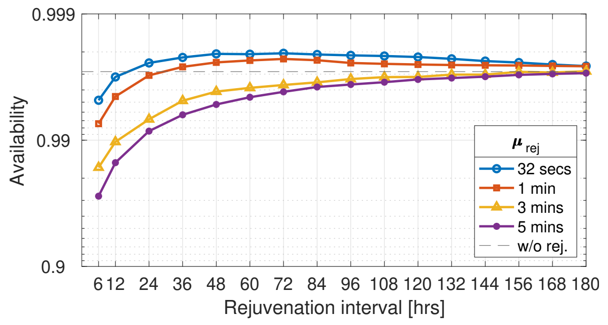
(a) *Manager*



(b) *Manager-Worker*



(c) *Multi-master*



(d) *Component-wise*

Fig. 10. Impact of rejuvenation policies on system availability for the different MANO deployments and for a varying rejuvenation duration.

In the *Multi-master* case, the overall availabilities are much higher and the maximum availability is achieved with a software restart every two days reaching 0.999915 with baseline parameters. Different to the other models, for short rejuvenation intervals, the difference between the rejuvenation durations (μ_{rej}) is less pronounced. This is because *Multi-master* entails a load-sharing cluster composed of three replicas of the MANO software which provides adequate protection even in cases where rejuvenation duration takes longer, i.e., longer downtime of one replica due to rejuvenation.

TABLE IV
STEADY-STATE AVAILABILITY WITH OPTIMAL REJUVENATION POLICY WHEN VARYING SOFTWARE AGING RATE.

Mean time to software aging	**Availability [Optimal rejuvenation interval in hrs]**			
	Manager	***Manager-Worker***	***Multi-master***	***Component-wise***
1 day	0.99665 [24]	0.99840 [36]	0.999626 [48]	0.99670 [36]
3 days	0.99743 [36]	0.99890 [48]	0.999816 [60]	0.99753 [36]
7 days	0.99762 [132]	0.999215 [96]	0.999915 [72]	0.99795 [132]
10 days	0.99791 [144]	0.999439 [108]	0.999956 [84]	0.99815 [156]

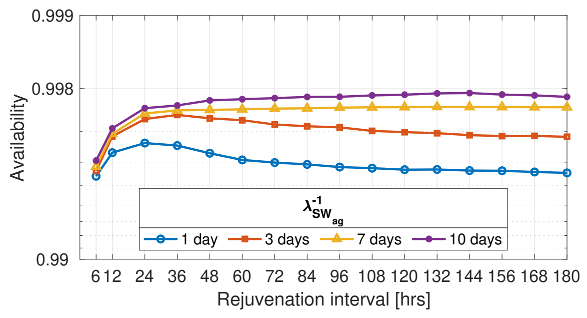
The results of the *Component-wise* model analysis show system performances that are much like the *Manager* model where for the baseline parameters the maximum achievable SSA differ at most $3.2 \cdot 10^{-4}$ compared to the *Manager* representation (0.99762 vs. 0.99794). Note that the rejuvenation schedules of the individual components are fully synchronized and the baseline duration equals 32 seconds, which is the highest amount of time required to restart the single components, i.e., LCM.

C. Software Aging Impact

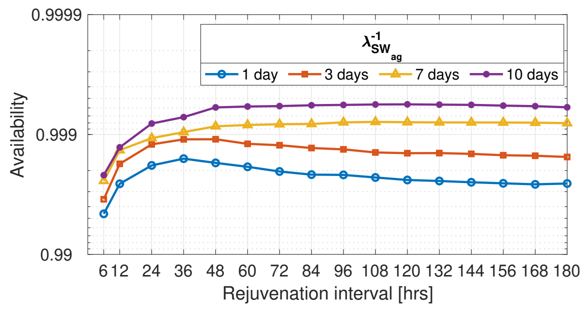
Software aging is an unpredictable parameter since it depends on several factors that may be out of developer's control such as software utilization rate, i.e., system load, operational profile and infrastructure, or software implementation. However, it has been shown that under high system workload, the software aging rate tends to increase, hence more aging errors are accumulated [67], [72], [73]. Consequently, the aging-related failure intensity increases.

We carry out a numerical analysis for a varying rejuvenation interval and assuming four software aging intensities representing high, medium, moderate, and low software utilization rates, i.e., 1, 3, 7 and 10 days mean time to software aging intensity. Fig. 11 illustrates the results for all the models. In addition, Table IV highlights the maximum availability figures for each of the deployments. Results reveal that for low to moderate software utilization, i.e., 7-10 days, the availability figures are closer compared to medium-high utilization. Such tendency is more evident as the rejuvenation interval decreases. Moreover, the difference between medium and high utilization is decreased when more fault-tolerance is introduced in the system, i.e., comparing the different MANO deployments. Again, the *Component-wise* system representation exhibits results almost identical to the *Manager*. For baseline parameters, the maximum SSA difference between the two models is within $3.3 \cdot 10^{-4}$ (0.99762 vs. 0.99795). Finally, for each of the case studies, the highest uptime gain is achieved for high software utilization indicating that a system under high workload can benefit more from rejuvenation.

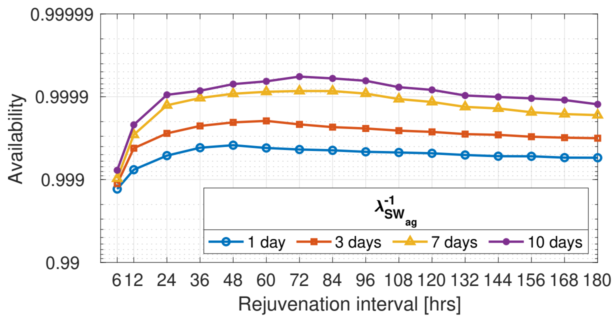
The sensitivity analysis revealed that aging failure rate may have a considerable impact on the availability of the MANO. We explore a range of software aging parameters by varying the aging rate and the aging failure rate between 1 and 10 days. Fig. 12 depicts the MANO availability for different parameter combinations, for each of the studied cases. In the *Manager* deployment case, it can be seen that the impact of the aging failure rates greatly depends on the



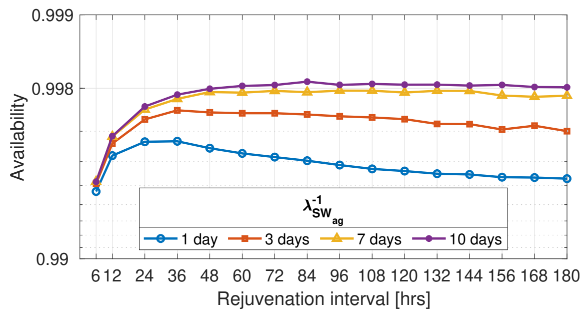
(a) *Manager*



(b) *Manager-Worker*

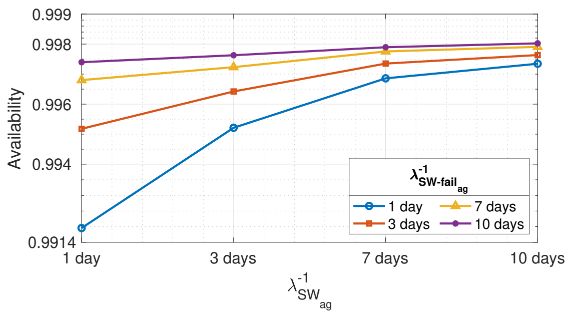


(c) *Multi-master*

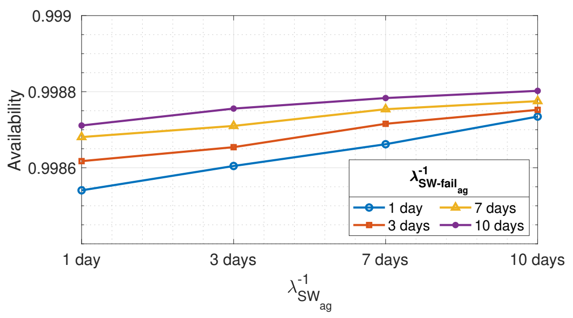


(d) *Component-wise*

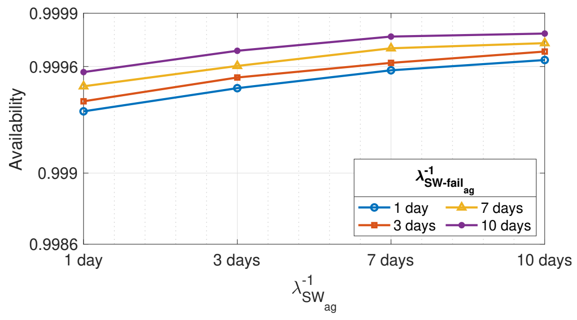
Fig. 11. System availability when varying rejuvenation interval and software aging rate.



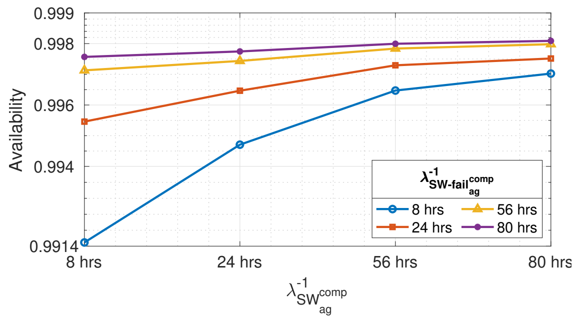
(a) *Manager*



(b) *Manager-Worker*



(c) *Multi-master*



(d) *Component-wise*

Fig. 12. Impact of software aging vs. aging-related failure.

rate of aging. For a short software aging time ($\lambda^{-1}_{SW_{ag}}$), i.e, lower than 7 days, an increase of the aging-related failure rate, i.e., lower time for software failure due to aging, can have a significant impact on the MANO availability. On the contrary, for a low to moderate software utilization, i.e., high times for the software to age, variations of the aging-related failure rates have a much lower impact. A similar trend, yet much less marked, is observed also for the *Manager-Worker* deployment. This tendency becomes negligible for the *Multi-master* deployment, therefore supporting the previous finding that adequate protection is needed on both host and software levels for neutralizing variations of critical parameters such as software aging and aging-induced failure rates.

Regarding the *Component-wise* model, Fig. 12(d) shows the sensitivity analysis for single components having three times lower failure intensities on the component's level compared to (such that the overall failure intensity of treating them together is the same as) the single MANO software model, i.e., *Manager*. Also for this analysis, the results show a trend very similar to the *Manager* model. The SSA of both models differs at most $5.05 \cdot 10^{-4}$. Moreover, increments or reductions of software aging parameters produce very much alike impacts for both models. Such observations, together with the previous insights, show that modeling of the MANO software as a single component yields a reasonable representative analysis of the system's steady state availability.

D. Threats to Validity

A possible limitation of this study concerns the precision in our numerical investigation. This is due to the accuracy of baseline parameters, which is, in general, common to many model-based studies. Although the majority of model parameters have been retrieved from related studies, we acknowledge that the choice of parameters may skew the analytic results. In particular, due to a lack of publicly available data regarding failure and recovery dynamics of MANO systems, we have made reasonable assumptions, based on studies regarding software of similar complexity. To lift this limitation a bit, we have performed experimental trials on a realistic MANO deployment aiming at retrieving recovery parameters' values of MANO software. Nevertheless, the very scope of the sensitivity analysis is to shed light onto the uncertainty related to these parameters, and two-orders of magnitude variation range is, in our opinion, sufficient to capture to a wide extent the uncertainties. An additional threat to the validity of our results is related to some assumptions regarding deployment configurations. In the *Multi-master* deployment, we assume that the load balancer is failure free. This is not the case for realistic deployments. However, from a deployment perspective, a service operator can limit the impact of this threat by using external load balancers which can provide a sufficient level of reliability. In addition, we also assume that while being hosted in the *Manager* node, regardless of the type of fault affecting the Worker node, the MANO software is only subject to non-aging related failures. While this does not reflect a realistic behavior, it is reasonable for those events that require a relatively short time to recover compared to the software aging rate. Overall, the goal of this work is to propose a methodology and model abstractions for assessing MANO implementations which can be used by system operators that have access to empirical data and can extract parameter values for use in the models.

IX. Conclusion

This paper presents four comprehensive availability models for a containerized NFV-MANO architecture encompassing various redundancy configurations. The models incorporate diverse failure modes and the corresponding recovery behaviors, regarding both hardware and software components. The models also include software aging effects and software rejuvenation, as proactive maintenance, aiming at mitigating aging effects. We performed an experimental campaign on real-life MANO system aiming at retrieving realistic system recovery parameters. We carried out an exhaustive sensitivity analysis from which we assess and quantify the steady-state availability and identified the impact that critical parameters have. The investigation showed that non-aging-related software failures and software repair rates stand out as key deteriorating failure and repair parameters, respectively. However, employing clustering mechanisms such as Kubernetes with redundancy on both host and software levels further boosts the NFV-MANO availability. Moreover, software aging can have a considerable impact on the MANO availability and we observed that a correct tuning of the rejuvenation policy can be beneficial and is particularly well-suited in cases where a high software utilization is experienced.

References

[1] ETSI, GS NFV, "ETSI GS NFV 001 v1. 1.1 Network Functions Virtualisation," *Use Cases. sl: ETSI*, 2013.

[2] G. Brown and H. Reading, "Service chaining in carrier networks," *Heavy Reading*, 2015.

[3] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7665.txt

[4] ETSI, GS NFV, "Network Functions Virtualisation (NFV): Architectural Framework," *ETSI GS NFV*, vol. 2, no. 2, p. V1, 2013.

[5] ETSI, "Network Functions Virtualisation, An Introduction, Benefits, Enablers, Challenges & Call for Action," *White Paper*, no. 1, pp. 1–16, 2012.

[6] ETSI, GS NFV, "Network functions virtualisation (NFV); management and orchestration," *NFV-MAN*, vol. 1, p. v0, 2014.

[7] R. Mijumbi *et al.*, "Management and orchestration challenges in network functions virtualization," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, 2016.

[8] ETSI, GS NFV, "ETSI GS NFV-REL 001 v1. 1.1: Network Functions Virtualisation (NFV); Resiliency Requirements," 2015.

[9] B. Han, V. Gopalakrishnan, G. Kathirvel, and A. Shaikh, "On the resiliency of virtual network functions," *IEEE Communications Magazine*, vol. 55, no. 7, pp. 152–157, 2017.

[10] B. Blanco *et al.*, "Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN," *Computer Standards & Interfaces*, vol. 54, pp. 216–228, 2017.

[11] K. S. Trivedi and A. Bobbio, *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.

[12] G. Arfaoui, J. M. Sanchez Vilchez, and J. Wary, "Security and Resilience in 5G: Current Challenges and Future Directions," in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 1010–1015.

[13] N. F. S. De Sousa, D. A. L. Perez, R. V. Rosa, M. A. Santos, and C. E. Rothenberg, "Network service orchestration: A survey," *Computer Communications*, vol. 142, pp. 69–94, 2019.

[14] A. J. Gonzalez *et al.*, "Dependability of the NFV orchestrator: State of the art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3307 – 3329, 2018.

[15] G. Nencioni *et al.*, "Orchestration and control in software-defined 5G networks: Research challenges," *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[16] ETSI, GS NFV, "ETSI GR NFV-REL 007 v1.1.2: Network Function Virtualisation (NFV); Reliability; Report on the resilience of NFV-MANO critical capabilities," 2017.

[17] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *Ieee Software*, vol. 33, no. 3, pp. 42–52, 2016.

[18] Docker Website. Accessed: 2020-11-15. [Online]. Available: "https://www.docker.com/"

[19] Linux Containers (LXC). Accessed: 2020-11-15. [Online]. Available: "https://linuxcontainers.org/"

[20] N. Dragoni *et al.*, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.

[21] T. Taleb, A. Ksentini, and B. Sericola, "On service resilience in cloud-native 5G mobile systems," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 483–496, 2016.

[22] S. Sharma, R. Miller, and A. Francini, "A Cloud-Native Approach to 5G Network Slicing," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 120–127, 2017.

[23] 5G-PPP Software Network Working Group *et al.*, "From webscale to telco, the cloud native journey," *Editor: Bessem Sayadi, July*, 2018.

[24] Open Source MANO (OSM). Accessed: 2020-11-15. [Online]. Available: "https://osm.etsi.org"

[25] OpenBaton. Accessed: 2020-11-15. [Online]. Available: "https://openbaton.github.io"

[26] SONATA website. Accessed: 2020-11-15. [Online]. Available: "https://www.sonata-nfv.eu/"

[27] ETSI, GS NFV, "ETSI GS NFV REL 004 v1.1.1: Network Functions Virtualisation (NFV); Assurance; Report on Active Monitoring and Failure Detection," 2016.

[28] ——, "Reliability; Report on Models and Features for End-to-End Reliability," no. GS REL 003 v1.1.2, 2016-07.

[29] A. Gonzalez *et al.*, "Service availability in the NFV virtualized evolved packet core," in *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[30] M. Di Mauro *et al.*, "IP multimedia subsystem in an NFV environment: availability evaluation and sensitivity analysis," in *2018 IEEE NFV-SDN*. IEEE, 2018, pp. 1–6.

[31] ——, "Service function chaining deployed in an NFV environment: An availability modeling," in *IEEE CSCN*. IEEE, 2017, pp. 42–47.

[32] ——, "Availability modeling and evaluation of a network service deployed via NFV," in *International Tyrrhenian Workshop on Digital Communication*. Springer, 2017, pp. 31–44.

[33] B. Tola, G. Nencioni, B. E. Helvik, and Y. Jiang, "Modeling and evaluating NFV-enabled network services under different availability modes," in *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*. IEEE, March 2019.

[34] T. Soenen *et al.*, "Optimising microservice-based reliable NFV management and orchestration architectures," in *The 9th International Workshop on Resilient Networks Design and Modeling*, Sep. 2017, pp. 1–7.

[35] S. Sebastio, R. Ghosh, and T. Mukherjee, "An availability analysis approach for deployment configurations of containers," *IEEE Transactions on Services Computing*, pp. 1–1, 2017.

[36] S. Sebastio, R. Ghosh, A. Gupta, and T. Mukherjee, "Contav: A tool to assess availability of container-based systems," in *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, 2018, pp. 25–32.

[37] M. Grottke and K. S. Trivedi, "Software faults, software aging and software rejuvenation," *The Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[38] K. S. Trivedi *et al.*, "Recovery from failures due to Mandelbugs in IT systems," *Proceedings of IEEE PRDC*, pp. 224–233, 2011.

[39] P. Vizarreta, K. Trivedi, V. Mendiratta, W. Kellerer, and C. Mas-Machuca, "DASON: Dependability Assessment Framework for Imperfect Distributed SDN Implementations," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 652–667, 2020.

[40] B. Tola, Y. Jiang, and B. E. Helvik, "On the Resilience of the NFV-MANO: An Availability Model of a Cloud-native Architecture," in *2020 16th International Conference on the Design of Reliable Communication Networks DRCN 2020*, 2020, pp. 1–7.

[41] L. Foundation. Open Network Automation Platform. Accessed: 2020-11-15. [Online]. Available: "https://www.onap.org/"

[42] G. N. ETSI, "Network functions virtualisation (NFV); management and orchestration; report on architectural options," vol. 1, p. v0, 2016.

[43] Juju Documentation. Accessed: 2020-11-15. [Online]. Available: "https://juju.is/docs"

[44] OpenStack Tacker. Accessed: 2020-11-15. [Online]. Available: "https://wiki.openstack.org/wiki/Tacker"

[45] Prometheus - Monitoring system and Time-series database. Accessed: 2020-02-26. [Online]. Available: "https://prometheus.io/docs/introduction/overview/"

[46] A. S. Foundation, "Apache Kafka," accessed: 2020-11-15. [Online]. Available: "\url{https://kafka.apache.org/intro}"

[47] Docker Documentation. Accessed: 2020-11-15. [Online]. Available: "https://docs.docker.com/engine/swarm/"

[48] Kubernetes Website. Accessed: 2020-11-15. [Online]. Available: "https://kubernetes.io/"

[49] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX-ATC 14)*, 2014, pp. 305–319.

[50] Creating Highly Available clusters with kubeadm. Accessed: 2020-11-15. [Online]. Available: "https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/"

[51] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007.

[52] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *2008 IEEE ISSRE Wksp*, Nov 2008, pp. 1–6.

[53] F. Machida, D. S. Kim, and K. S. Trivedi, "Modeling and analysis of software rejuvenation in a server virtualized system with live VM migration," *Performance Evaluation*, vol. 70, no. 3, pp. 212–230, 2013.

[54] M. Escheikh, Z. Tayachi, and K. Barkaoui, "Workload-dependent software aging impact on performance and energy consumption in server virtualized systems," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016, pp. 111–118.

[55] M. Torquato and M. Vieira, "Interacting SRN models for availability evaluation of VM migration as rejuvenation on a system under varying workload," in *2018 IEEE International symposium on software reliability engineering workshops (ISSREW)*. IEEE, 2018, pp. 300–307.

[56] E. Guedes and P. Maciel, "Stochastic model for availability analysis of service function chains using rejuvenation and live migration," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 211–217.

[57] P. Vizarreta, P. Heegaard, B. Helvik, W. Kellerer, and C. M. Machuca, "Characterization of failure dynamics in SDN controllers," in *2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM)*, 2017, pp. 1–7.

[58] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: Formal definitions and concepts," in *School organized by the European Educational Forum*. Springer, 2000, pp. 315–343.

[59] Möbius: Model-based environment for validation of system reliability, availability, security and performance. Accessed: 2020-11-15. [Online]. Available: "https://www.mobius.illinois.edu"

[60] T. A. Nguyen, D. S. Kim, and J. S. Park, "A comprehensive availability modeling and analysis of a virtualized servers system using stochastic reward nets," *The Scientific World Journal*, vol. 2014, 2014.

[61] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, 1991.

[62] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 249–265.

[63] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 58–72.

[64] R. Hanmer, L. Jagadeesan, V. Mendiratta, and H. Zhang, "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)*. IEEE, 2018, pp. 59–64.

[65] G. M. Yilma, Z. F. Yousaf, V. Sciancalepore, and X. Costa-Perez, "Benchmarking open source NFV MANO systems: OSM and ONAP," *Computer Communications*, vol. 161, pp. 86–98, 2020.

[66] OSM Bugzilla bug tracking system. Accessed: 2020-11-15. [Online]. Available: "https://osm.etsi.org/bugzilla/describecomponents.cgi"

[67] M. Torquato and M. Vieira, "An experimental study of software aging and rejuvenation in dockerd," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 1–6.

[68] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, p. 73–88, Oct. 2001. [Online]. Available: https://doi.org/10.1145/502059.502042

[69] K. Kanoun and Y. Crouzet, "Dependability benchmarks for operating systems," *International Journal of Performability Engineering*, vol. 2, pp. 275–287, 2006.

[70] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the linux operating system," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 71–80.

[71] D. Cotroneo, A. K. Iannillo, R. Natella, and S. Rosiello, "Dependability assessment of the Android OS through fault injection," *IEEE Transactions on Reliability*, 2019.

[72] R. Matos, J. Araujo, V. Alves, and P. Maciel, "Characterization of software aging effects in elastic storage mechanisms for private clouds," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, 2012, pp. 293–298.

[73] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono, "Software aging analysis of the android mobile OS," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 478–489.

Besmir Tola received the M.Sc. degree in Electronics and Telecommunication Engineering from the University of Siena (Italy) in 2014. In autumn 2015, he joined the IIK department at the Norwegian University of Science and Technology (NTNU) as a Ph.D. candidate in Information Security and Communication Technology. From spring 2020, he holds an Assistant Professor position within the same department. In 2016 and 2018, he was a visiting researcher at the Nokia Bell Labs in Stuttgart (Germany), and UNINETT (Norwegian National Research and Education Network Operator), respectively, where he worked on dependability modeling and analysis of cloud computing infrastructures and services. His research interests include performance and dependability analysis of cloud computing, SDN, and NFV architectures.

Yuming Jiang received the B.Sc. degree from Peking University and the Ph.D. degree from the National University of Singapore. He has been a Professor with the Norwegian University of Science and Technology, Trondheim, Norway, since 2005. From 1996 to 1997, he was with Motorola, Beijing, China, and from 2001 to 2003, he was with the Institute for Infocomm Research (I2R), Singapore. He visited Northwestern University from 2009 to 2010, and Columbia University from 2015 to 2016. His research interests are the provision, analysis, and management of quality of service guarantees, with a focus on (stochastic) network calculus and its applications. He has authored the book entitled Stochastic Network Calculus. He was a Co-Chair of IEEE Globecom 2005 - General Conference Symposium, a TPC Co-Chair of 67th IEEE Vehicular Technology Conference (VTC) 2008, the General Chair of IFIP Networking 2014 Conference, the Chair of the 2018 International Workshop on Network Calculus and Applications, and a TPC Co-Chair of the 32nd International Teletraffic Congress (ITC32), 2020.

Bjarne E. Helvik (1952) received his Siv.ing. degree (MSc in technology) from the Norwegian Institute of Technology (NTH), Trondheim, Norway in 1975. He was awarded the degree Dr. Techn. from NTH in 1982. He has since 1997 been Professor at the Norwegian University of Science and Technology (NTNU), the Department of Telematics and Department of information Security and Communication Technology. In the period 2009 – 2017, he has been Vice Dean with responsibility for research at the Faculty of Information Technology and Electrical Engineering at NTNU. He has previously held various positions at ELAB and SINTEF Telecom and Informatics. In the period 1988-1997 he was appointed as Adjunct Professor at the Department of Computer Engineering and Telematics at NTH. During 2003 - 2012 Principal investigator at the Norwegian Centre of Excellence Q2S - the Centre for Quantifiable Quality of Service and is since 2020 Principal investigator at the Centre for Research based Innovation NORCICS - Norwegian Center for Cybersecurity in Critical Sectors. His field of interests includes QoS, dependability modelling, measurements, analysis and simulation, fault-tolerant computing systems and survivable networks, as well as related system architectural issues. His current research is on ensuring dependability in services provided by multi-domain, virtualised ICT systems, with activities focusing on 5G and SmartGrids.