UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEPENDABILITY BENCHMARKING OF NETWORK FUNCTION VIRTUALIZATION

LUIGI DE SIMONE

TUTOR: PROF. DOMENICO COTRONEO

XXIX CICLO

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

Abstract

Network Function Virtualization (NFV) is an emerging networking paradigm that aims to reduce costs and time-to-market, improve manageability, and foster competition and innovative services. NFV exploits virtualization and cloud computing technologies to turn physical network functions into *Virtualized Network Functions* (VNFs), which will be implemented in software, and will run as Virtual Machines (VMs) on commodity hardware located in high-performance data centers, namely *Network Function Virtualization Infrastructures* (NFVIs). The NFV paradigm relies on cloud computing and virtualization technologies to provide *carrier-grade* services, i.e., the ability of a service to be highly reliable and available, within fast and automatic failure recovery mechanisms. The availability of many virtualization solutions for NFV poses the question on which virtualization technology should be adopted for NFV, in order to fulfill the requirements described above. Currently, there are limited solutions for analyzing, in quantitative terms, the performance and reliability tradeoffs, which are important concerns for the adoption of NFV. This thesis deals with assessment of the reliability and of the performance of NFV systems. It proposes a methodology, which includes context, measures, and faultloads, to conduct dependability benchmarks in NFV, according to the general principles of dependability benchmarking. To this aim, a fault injection framework for the virtualization technologies has been designed and implemented for the virtualized technologies being used as case studies in this thesis. This framework is successfully used to conduct an extensive experimental campaign, where we compare two candidate virtualization technologies for NFV adoption: the commercial, hypervisor-based virtualization platform *VMware vSphere*, and the open-source, container-based virtualization platform *Docker*. These technologies are assessed in the context of a high-availability, NFV-oriented *IP Multimedia Subsystem*

(IMS). The analysis of experimental results reveal that i) fault management mechanisms are crucial in NFV, in order to provide accurate failure detection and start the subsequent failover actions, and ii) fault injection proves to be valuable way to introduce uncommon scenarios in the NFVI, which can be fundamental to provide a high reliable service in production.

Acknowledgements

I would like to thank my advisor Prof. Domenico Cotroneo. He believed in me and supported me to pursue the PhD.
Thanks to Prof. Russo that gives me valuable tips especially during the thesis period.
Thanks to Prof. Neeraj Suri and all DEEDS guys! I would like to thank in particular Stefan. He was my guardian angel (in every senses) during my visiting in Darmstadt.
Thanks to Prof. Peter Popov and Prof. Henrique Maderia to revise my thesis and give me precious comments.
Thanks to my (practically :D) second advisor Roberto. During this years he taught me the most of scientific things I know today. Thanks to all my colleagues, Antonio, Roberto P., Stefano, Ken, Fabio, Raffo, Mario, Anna, Flavio, Salvatore, Alma (also if you are not an eng XD). They were been really part of me during my PhD program.
Thanks to my family because the man who I am now is only due to them, my father, my mother, my bro, and my sisters. Thanks to my second family where I've always been a son.
Thanks to my sisters and brothers (in God :D) because they supported me, by crying and smiling in all situations.
Thanks to love of my life, Ilaria, because without her I don't know if I would still be alive today. She always been there for me, really for better, for worse, for richer, for poorer, in sickness and health. I will love you until death do us part!!!
It seemed impossible to me, but I did it...my (crazy, of course) way :D
I want to finish as I finished the acknowledgements in my MSc thesis.
I hope I'll be ok,I hope I'll succeed! ("Io speriamo che me la cavo!")

Naples, Italy, 10th April 2017 Luigi De Simone

Contents

This page intentionally left blank.

List of Acronyms

The following acronyms are used throughout this text.

CCE	Cloud Computing Ecosystem
COTS	Commercial Off-The-Shelf
ETSI	European Telecommunications Standards Institute
FTAM	Fault Tolerance Algorithms and Mechanism
HBA	Host Bus Adapter
MANO	Management and Orchestration
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NIC	Network Interface Controller
OS	Operating System
PM	Physical Machine
VM	Virtual Machine

VMFS	VMware Virtual Machine File System
VNF	Virtual Network Function

List of Tables

This page intentionally left blank.

List of Figures

Chapter 1

Dependability in NFV: Challenges and Contributions

1.1 Dependability in Network Function Virtualization

Network Function Virtualization (NFV) [111, 112] is an emerging networking paradigm that aims to reduce costs and time-to-market, improve manageability, and foster competition and innovative services [112, 96]. NFV will take advantage of virtualization and cloud computing technologies to turn network functions (such as IMS, EPC, DPI, NAC, etc.) into *Virtualized Network Functions* (VNFs), which will be implemented in software, and will run as Virtual Machines (VMs) on commodity hardware located in high-performance data centers, namely *Network Function Virtualization Infrastructures* (NFVIs).

Despite its recent introduction, NFV has already gained significant market traction. A recent report [147] shows that the global NFV market is growing at a CAGR of 32.88% during the period 2016-2020. Furthermore, there is an increasing offer and a high number of NFV vendors that compete on VNF, NFV management and orchestration (MANO), and network acceleration products [134, 104]. Moreover, in the foreseeable future, VNFs and NFVIs will be provided on a pay-per-use basis, according to the *as-a-service* cloud business model.

NFV solutions have to compete not only in cost and manageability, but also in performance and reliability: telecom regulations impose *carrier-grade* requirements to network functions, which need to achieve extremely low packet processing overheads, controlled latency, and efficient virtual switching, along with quick and automatic recovery from faults (in the order of few seconds) and extremely high availability (99.999% or higher) [130]. It is well known that these requirements are well satisfied by traditional (hardware-based) network functions, which have been proven very reliable over the last decades. However, performance and reliability is definitively a big challenge to achieve in next generation of network functions, where most of the control logic will be implemented by means of software and virtualization technologies.

Figure 1.1 briefly shows the architecture of an NFV system. It is characterized by three kinds of components.

- **Virtualized Network Functions (VNFs)** are network functions implemented in software. VNF are used to process the network traffic (according to the specific network protocol and network topology). They use both virtual and physical resources. Furthermore, the Element Management (EM) components perform management functions for a specific VNF (such as monitoring, configuring, etc.);

- **NFV Infrastructure (NFVI)** abstracts and manages access to physical resources. It includes the hardware resources, a virtualization layer to create virtual resources on the available hardware, and

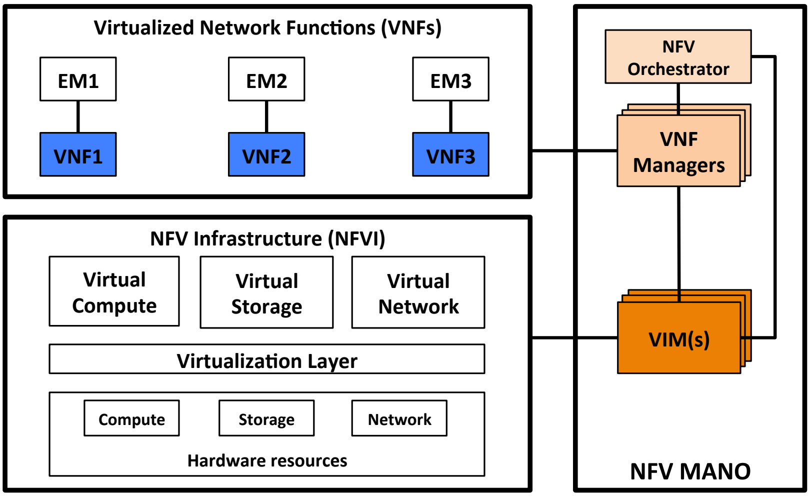


Figure 1.1. NFV architecture.

the virtual resources themselves[1];

- **NFV Management and Orchestration (MANO)** acts as coordinator of the overall NFV system. It includes three types of subcomponents: an *Orchestrator*, which allocates and releases resources of the NFVI to the VNFs, by using the VIM, and manages the lifecycle of network services (creation, scaling, configuration, upgrading, termination); *VNF managers*, which are used to manage the lifecycle of VNFs. Each VNF is linked to a VNF manager; the *Virtualized Infrastructure Managers (VIMs)*, which are controlled by the NFV Orchestrator and VNF Managers to manage physical and virtual resources in the NFVI.

The NFV paradigm relies on cloud and virtualization technologies, which add more complexity and new risks: since these technologies are

[1] In the context of virtualization, the term *Virtual Machine* generally refers to an isolated domain with abstract resources. However, the ETSI NFV framework does not restrict the application of NFV to a particular virtualization technology [119].

not still well studied and understood in the context of NFV, they may represent a threat for the performance and reliability of network infrastructures. It is well known that telecom services are expected to be highly available and, as soon as a failure or an outage occurs, they must be recovered within a short period of time (e.g., milliseconds), using automatic recovery means. Therefore, achieving dependability in NFV has been recognized as a big challenging issue, and is currently being discussed by the European Telecommunications Standards Institute (ETSI) within the *resiliency*[2] term. **ETSI GS NFV-REL** (2015) [115] identifies use cases, requirements and architectures that will serve as a reference for the emerging NFV technologies, including *resiliency requirements* that the emerging NFV architectures will have to meet.

In this document, the ETSI addresses the problem of NFV resiliency by recommending design practices, including failure detection and isolation, automated recovery from failures, prevention of single points of failure in the architecture, and so on. The document also introduces *service continuity* as the capability of assuring quality of service goals when anomaly conditions, such as failure ones, occur. NFV systems have to assure service continuity even if a component fails, or if the workload is much higher than the nominal one. Another aspect of resiliency is the *service availability level* of network functions. The VNFs must guarantee that the provided quality of service is the same as hardware-based network functions (i.e., legacy networks). In order to meet these objectives, the ETSI NFV architecture will include resiliency mechanisms both at the VNF layer and at the NFV-MANO (Management and Orchestration) layer [116]. Finally, the ETSI NFV use cases envision the adoption of both stateless and stateful VNFs. In the former case, VNF instances can be scaled to accommodate high volumes of traffic, and to recover from failures. As for stateful VNFs, they

[2] Resiliency is the capability of a system to adapt itself properly when facing faults or changes in the environment [89]

will require mechanisms for storing and recovering the state of network sessions and connections in a reliable way.

In particular, the ETSI identifies two main conditions that can cause a service to deviate from normal operation: congestion conditions and failure conditions. *Congestion conditions* occur when an unusual volume of traffic saturates the capacity of VNFs. This situation may result from special events (e.g., Chinese New Year festival, TV shows, etc.) or from a cyber-attacks (e.g., DDoS attack). Instead, during *failure conditions*, a service may be interrupted or it becomes unavailable due to faulty components.

In these situations, two key factors play a crucial role: priority of restoration and failure recovery time. The *priority of restoration* level denotes which is the service that has the main impact on the NFVI availability as a whole. Thus, restoring a service with a high priority increases the overall service availability. The *failure recovery time* is the time needed to recovery from failures, and it depends both on the amount of redundant resources available and effectiveness of fault management mechanisms implemented. Furthermore, it is worth noting the real-time nature of network functions: a latency-sensitive service need to be recovered as fast as possible, rather than a lower priority service in which the availability level is not stringent.

It is clear that all these requirements introduce the need for fair and accurate procedures to *benchmark* both the quality of service and the reliability level of competing NFV solutions and providers.

1.2 Dependability threats in NFV scenarios

The NFV paradigm relies on cloud computing and virtualization technologies to provide carrier-grade services. As mentioned before, in the context of telecommunication domain, *carrier-grade* means the ability of a service to be highly reliable and available, well tested, and with proven capabilities. In this section, we discuss which dependability threats impact NFV-based telecommunication services.

Cloud computing was born to avoid the need for many corporations to build and manage their own IT data centers. Cloud paradigm is meant to provide computational resources, in such a way to allocate and deallocate them on demand, based on a pay-per-use business model, avoiding expensive hardware platforms and big initial capital costs. Such computing resources include Internet services, storage facilities, compute power, all provided just like a service. In the last years, both industry and researchers are debating about how much we can trust in cloud computing [21, 103]. Recently, several cases of cloud failures such as Amazon Web Services (AWS) [11] (a race condition in the code on the servers which manage data storage), Microsoft Azure [62] (a network device misconfiguration) and Google Docs [161] (software fault in the memory management component), have raised concerns on cloud computing. From the cloud provider perspective, failures can lead to billion of money losses, just for an hour of offline time [42]. These outages result from faults that occur in a component of the cloud system, and that propagate through the entire ecosystem. Thus, delivering a *trustworthy* cloud computing service (ranging from IaaS to SaaS [97]) is a priority. In general, many organizations and companies rely on cloud computing services (e.g., in traffic management system for decision support [92], in finance [2], in healthcare [3], etc.), therefore, cloud services will become more critical in the near future.

Currently, there is a wide spectrum of approaches in the virtualization panorama, but mainly *Hypervisor-based* and *Container-based* virtualization are the two main solutions actually in use in cloud computing infrastructures [64, 10].

Hypervisor-based virtualization technologies provide a full emulation of a computer machine. It allows guest operating systems to run on top of a so-called hypervisor, which is a piece of software that manages multiple virtual machines on top of a single physical machine. Examples of hypervisors are VMware ESXi [155], KVM [86], Microsoft Hyper-V [102], Xen [17].

On the other side of spectrum, **Container-based** virtualization, also called *Operating System-level* virtualization, allows to run multiple instances of an OS, but without the overhead of fully emulating computer machines. The idea is to enhance the traditional system call interface of OSes, in order to provide isolated environments (*containers*) to applications. A container is not a virtual machine in the traditional sense, but it is an environment with its own filesystem, network interfaces, processes, etc., by leveraging on resource abstraction mechanisms (e.g., *namespaces* in Linux [94]) and resource management capabilities (e.g., *cgroups* in Linux [98]) provided by the kernel of the OS. Examples of container-based virtualization technologies are LXC (LinuX Container [95]), Docker [49], OpenVZ [122].

The availability of many virtualization solutions for NFV architects poses the question on which virtualization technology should be adopted for NFV. On the one hand, container-based virtualization is perceived as a promising solution to reduce the overhead of virtualization, and thus to achieve higher performance and scalability. On the other hand, traditional hypervisor-based virtualization enforces a stronger isolation among virtual machines and the physical machines, and it is regarded as a more secure and reliable solution [138]. However, there are limited solutions (as will be shown in the discussion of related work, see Section 2.2) for analyzing, in quantitative terms, performance and reliability trade-offs. This lack is especially problematic in the case of NFV, where both performance and reliability aspects are important concerns.

The problem of testing and validating cloud computing infrastructures is a big challenge. Since a cloud computing infrastructure consists of many elements, it is hard to understand how individual elements impact on the reliability of the cloud service/infrastructure, and how to prevent failures [47]. In the context of NFV, telecom operators are concerned by the availability of their products and the user-perceived dependability because (i) unreliable services are likely to be discarded by users and (ii) the total costs of system failures can be tremendous. Potential causes of failures in NFV are:

- *Hardware Faults*: the use of Commercial Off-The-Shelf (COTS) hardware (instead of application-specific hardware equipment) is a potential source of faults in the NFVI, since these hardware components have not been designed for carrier-grade requirements (e.g., in terms of time-to-failure of hardware components). Ideally, these faults can be masked by using redundant hardware components and by taking advantage of this redundancy in the virtualization layer (e.g., the virtualization layer can mask physical disk faults by replicating data on several physical disks, and by exposing only one, reliable virtual disk to the business layer);

- *Software Faults*: NFVI will also reuse COTS software components at various levels of the virtualization stack, including the hypervisor, guest OSes, management and orchestration software, middleware, and third-party VNF software; however, COTS software is not subject to rigorous development and testing processes, thus these components are often plagued by residual software bugs [108]. Moreover, software for NFVI is tightly related to storage and network devices. These features expose VNFs which is running on top of NFVIs to robustness and performance issues;

- *Operator Faults*: the complexity and the large scale of virtualization infrastructures expose them to human mistakes during operations and configuration. Since these infrastructures often involve thousands of virtual nodes, and since they have complex network topologies and software/hardware stacks, it is difficult for system administrators to configure and to operate them, and to make an accurate capacity plan; thus, these infrastructures are exposed to mistakes that may happen during deployment and operation.

1.3 Dependability Benchmarking in NFV

Benchmarks are an established practice for performance evaluation in the computer industry since decades. Examples of successful benchmarking initiatives are the TPC (Transaction Processing Performance Council [149]) and the SPEC (Standard Performance Evaluation Corporation [141]).

More recently, the research community developed *dependability benchmarking* procedures, which have significantly matured from both the methodological and from technical point of view [55, 87, 54, 153]. However, dependability benchmarking is a more difficult task than performance and functional benchmarking, as it needs to consider the presence of faults in the system, which requires elaborated test scenarios and experimental procedures, by leveraging on dependability evaluation techniques (in particular, fault injection).

The goal of dependability benchmarking is to measure the dependability properties of a computer system or component, in an automated, reproducible, portable, and trustworthy way [106, 82]. The ultimate aim is to enable system designers to make informed purchase decisions about different system's and/or component's choice. Dependability were successfully adopted in the context of COTS (Commercial Off The Shelf) components, such as Operating Systems and Databases. A dependability benchmark is aimed at addressing the interests (such as fairness and ease of applicability) of several stakeholders, including product manufacturers and users. To be accepted both by the computer industry and by the user community, the benchmark must state the measures, the procedure and conditions under which the measures are obtained, and the domain in which these measures are considered valid and meaningful. The procedures and rules to be followed have to be specified in detail to enable users to implement the benchmark for a given system and to interpret the results.

The *DBench* project [45, 82] proposed the first general framework for dependability benchmarking. DBench defined dependability benchmarks for several types of product, such as DBMSs, OSes, and web-servers.

Dependability benchmarking becomes more compelling for NFV, as denoted by the interest of standardization bodies to define reliability requirements and evaluation procedures for the cloud and for NFV [57, 4], and also the effort to drive the consistent implementation of an open and standard NFV reference platform [128]. As mentioned, the need for dependability benchmarking is exacerbated by the high incidence of the faults, due to the large scale and complexity of NFVIs, the dynamism of NFV services, and the massive adoption of commercial off-the-shelf (COTS) hardware and software components. While COTS components are easily procured and replaceable, NFV will need to recover from faulty components in a timely way and preserve high network performance. However, there is a lack of approaches that could allow NFV vendors, providers and users to evaluate dependability, with a degree of accuracy and trustworthiness comparable to performance benchmarks.

1.4 Thesis Contributions

In the light of the context and of the motivations discussed before, this thesis dissertation contributes towards assessing the reliability and performance of NFV systems, paving the way for the creation of a general Dependability Benchmark framework. More specifically, the contributions of this thesis include the following points.

- **Definition of a dependability benchmarking methodology for NFV (Cap. 3).**

  This dissertation provides context, measures, and faultloads to conduct dependability benchmarks in NFV, according to the general principles of dependability benchmarking, i.e., representative, simple, and portable across alternative technologies [44]. Moreover, the benchmark takes into account the NFV use cases and technologies. The benchmark allows to: (i) get quantitative measures of worst-case quality of service; (ii) identify which fault types and faulty components impact most on NFV services; (iii) validate the effectiveness of fault tolerance and high-availability algorithms and mechanisms.

  The faultload is obtained by modeling the virtualized infrastructure for the four domains according to a typical virtualized infrastructure: *network*, *storage*, *CPU*, and *memory*. These elements are present both as *virtual* resource abstractions, and as *physical* resources. The fault model is aimed to the benchmark performer, which builds the faultload for the target infrastructure by systematically applying the fault model to each resource in the infrastructure (virtual and physical machines, virtual and physical disks and switches). For each domain (physical and virtual CPU, memory, disk and network), we identify faults according to the following three general fault types: *unavailability* (the resource becomes unresponsive and unusable); *delay* (the resource is overcommitted and slowed-down); *corruption* (the information stored or processed by the resource is invalid). These fault types are broad classes that span over the possible failure modes of a component [126, 106, 19, 38, 73]. We specialize these general fault types for each resource, by analyzing how hardware, software, and/or operator faults can likely cause these three possible fault types [44]. In this analysis, we consider the scientific literature on fault injection and failure analysis in cloud computing infrastructures ([79], [67], [77], [125], [91], [90], [12]), well-known cloud computing incidents ([11] , [62], [161]), and knowledge on the prospective architecture and products for NFVI [154], to identify a representative and complete set of faults.

- **A Fault Injection toolsuite for virtualization technologies (Chap. 4)**. Dependability benchmarking requires fault injection testing technologies to support the experimental evaluation. Therefore, this dissertation presents the design and implementation of fault injection tools for both VMware vSphere and Docker, by adopting the fault model defined by the dependability benchmarking methodology. The faults are injected by emulating their effects on the virtualization layer. In particular, I/O and compute faults can be emulated, respectively, by deliberately injecting I/O losses, corruptions and delays, and by injecting code and data corruptions in memory and in CPU registers, by forcing a crash of VMs and of their hosting nodes, and by introducing CPU- and memory-bound "hogs" (that is, tasks that deliberately consume CPU cycles and allocate memory areas) in order to cause resource exhaustion. These faults can be injected either in a specific virtual domain (a VM or a container), or in a physical machine (PM) of the NFVI.

- **A NFV case study on dependability benchmarking (Chap. 5)**. This dissertation presents an experimental case study to compare, using the proposed dependability benchmark, two candidate virtualization technologies for NFV: the commercial, hypervisor-based virtualization platform *VMware vSphere*, and the open-source, container-based virtualization platform *Docker*. We evaluate these technologies in the context of a high-availability, NFV-oriented *IP Multimedia Subsystem* (IMS), which has been deployed on two alternative NFVI configurations.

Chapter 2

Related Work

2.1 Dependability Benchmarking

The goal of dependability benchmarking is to quantify the dependability properties of a computer system or component, in an automated, reproducible, portable and trustworthy way [106, 82]. This goal is especially important for COTS components and COTS-based systems, since it would enable system designers to make informed purchase decisions. While there are well-established benchmarks for functional and performance features, the benchmarking of dependability is a more difficult task as it needs to consider the presence of faults in the system, thus requiring more sophisticated test scenarios and experimental procedures, by leveraging on dependability evaluation techniques (in particular, fault injection). A dependability benchmark is aimed at addressing the interests (such as fairness and ease of applicability) of several stakeholders, including product manufacturers and users. To be accepted both by the computer industry and by the user community, the benchmark must state the measures, the procedure and conditions under which the measures are obtained, and the domain in which these measures are considered valid and meaningful. The procedures and rules to be followed have to be specified in detail to enable users to implement the benchmark for a given system and to interpret the results.

The following section presents the necessary background to understand all the specifics of dependability benchmarking, as well as all the prior work on this subject.

2.1.1 Basic Concepts

The *DBench* project [45, 82] proposed a general framework for dependability benchmarking, and defined dependability benchmarks for several types of product, such as DBMSs, OSes, and web-servers. In order to rigorously define a benchmarking methodology for NFV, we briefly review the basic elements of this framework.

A dependability benchmark distinguishes between the **Benchmark Target** (BT), which is the component or system under evaluation, and the **System Under Benchmark** (SUB), which includes the BT along with other resources (including hardware and software) that are needed to run the BT. The evaluation process is driven by the **benchmark context**, which identifies the *benchmark user* (which takes advantage of the results) and the *benchmark performer* (which carries out the benchmark experiments), the *life cycle phase* of the benchmark target at the time of the benchmark (such as, the system integration phase), and the benchmark purpose (such as, identifying weak points during the testing phase, or to qualitatively/quantitatively validate dependability features supported or claimed). Finally, the benchmark defines **measures** that reflect the dependability and performance of the BT, either qualitatively (e.g., in terms supported fault-tolerance capabilities) or quantitatively (e.g., system availability and response time in the presence of faults). Moreover, the measures can be either *comprehensive* (e.g., reflect the quality of service of the system as a whole) or *specific* (e.g., related to a specific fault-tolerance capability).

Dependability benchmarks apply two forms of stimuli on the system, namely the **workload** and the **faultload**. The workload represents the typical operational profile for the considered system, and it is defined according to similar considerations that are made for classical performance benchmarks [76, 139, 148]. The faultload is a peculiarity of dependability benchmarks: it defines a set of faults and exceptional conditions, which are injected to emulate the real threats that the system is expected to face. The definition of a realistic faultload is probably the most difficult part of defining a dependability benchmark, as it requires a pragmatic process based on knowledge, observation, and reasoning [43]. The most important source of information is the post-mortem analysis of failures in operational systems (e.g., running previous versions of the product, or similar products). These data can be gathered from empirical studies, or it can be directly collected from end-users and service providers. Alternatively, the faultload can be identified from a systematic analysis of system's components and their potential faults, based on expert judgment. A common approach is to define selective faultloads, each addressing one class of faults: hardware, software, and operator faults [66, 143, 54, 24, 123, 153].

It is important to remark that dependability benchmarks separate the BT from the so-called **Fault Injection Target** (FIT), that is, the component subject to the injection of faults. This separation is important since it is desirable not to modify the BT when applying the faultload, in order to get the benchmark accepted by its stakeholders. For this reason, fault injection introduces perturbations outside the BT. Moreover, this approach allows to compare several BTs with respect to the same set of faults, since the injected faults are independent from the specific BT. Figure 3.2 summarizes the relationship between elements described before.

Once these elements are defined, a dependability benchmark entails the execution of a sequence of fault injection experiments. In each fault

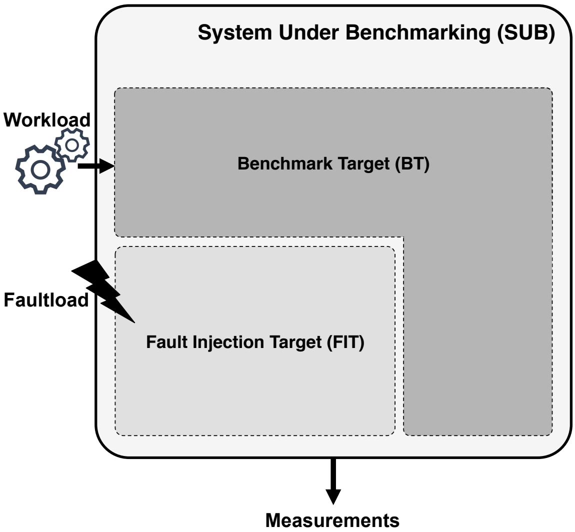


Figure 2.1. Benchmark Target (BT), System Under Benchmark (SUB), Fault Injection Target (FIT), Faultload, Workload, and Measurements relationship.

injection experiment, the SUB is first deployed and configured; then, the workload is submitted to the SUB and, during its execution, faults from the faultload are injected; at the end of the execution, performance and failure data are collected from the SUB, and the experimental testbed is cleaned-up before starting the next experiment. This process is repeated several times, by injecting a different fault at each fault injection experiment, while using the same workload and collecting the same performance and failure data. These data are processed after the experimental campaign, to compute the dependability measures of the benchmark. The execution of fault injection experiments is typically supported and automated by tools for test infrastructure management, workload generation, fault injection, and monitoring [72, 162, 109].

After running the dependability benchmark, we must assure that the results are meaningful, relevant and can be shared among various benchmark users (e.g., computer industry, researcher, practioners) without any mystification. As mentioned in Sec. 1.3, the benchmark must fulfill different key objectives including:

- *Representativeness*, which embraces the measures, the faultload, and the workload ability to be representative regarding the system under test. The definition of the measures have to be done thinking about the real usefulness for the benchmark users. The workload must include realistic profiles that we found in real systems during operations. Finally, the faultload must consider the real threats that the system can experience during operations. In general, the representativeness strongly impacts on the relevance of the results;

- *Repeatability and Reproducibility. Repeatability* is the property of the benchmark to show statistically equivalent results when it is performed several time in the same environment (i.e, adopting the same workload, faultload, SUB). That property is fundamental to guarantee the trustworthiness of the benchmark results. *Reproducibility* instead regards the implementation from another party of a benchmark from the given specifications;

- *Portability*, which includes the ability of the benchmark to run on different target systems, allowing comparison of different systems or their components. The portability of the benchmark heavily depends on the portability of the faultload;

- *Non-intrusiveness*, which reflects the implementation of the benchmark and the changes (if needed) in the system under benchmark. Naturally, it would be desirable minimum changes in the SUB to make the benchmark less intrusive as possible;

- *Scalability*, which is the property related to the ability of the benchmark to be applied on systems of different sizes. In particular, this property affects the workload in terms of number of inputs submitted to the SUB; however, the scalability is also related to the faultload because for larger systems the number of components that can fail simultaneously increase, thus the number of faults gets higher;

2.1.2 Dependability Benchmarking studies

Several dependability benchmarks have been proposed to compare alternative components, such as operating systems [55, 87], web servers [54], and DBMSs [153].

In [87, 55, 8], UNIX and Windows OSs were compared with respect to the *severity of their failures*: for each experiment, the behavior of the OS has been ranked according to a failure severity scale, to reflect the impact of the fault on the stability and responsiveness of the system. The measures provided are related to the *OS robustness* against failing system calls, the *OS reaction time* for faulty system calls, and the *OS restart time*, that is the time needed to the OS to reboot after a faulty system calls is activated. Finally, they consider three different workloads: (i) an application that implements the experiments control system of the TPC-C performance benchmark [148], (ii) the PostMark [85] file system performance benchmark for operating systems, and (iii) the Java Virtual Machine (JVM). Regarding the fault load, it is based on the corruption of systems call parameters. It is worth to mention another study [53] which characterize the operating system (in particular they target the Windows operating system family) in the presence of software faults in OS components.

In [54], the SPECweb benchmark [139] has been extended to evaluate web servers with respect to throughput, response time, error rate, autonomy and availability in the presence of *software faults* ("bugs") injected in the OS. Similarly, in [153], the TPC-C benchmark [148] has been extended to evaluate OLTP products and configurations with respect to number of transactions performed per minute and availability, in the presence of *operator faults* (e.g., faults affecting the database schema or the filesystem). More recently, the *ISO/IEC Systems and software Quality Requirements and Evaluation* (SQuaRE) standard [1] defined an evaluation module (*ISO/IEC 25045*) that integrates the concepts from dependability benchmarking [58] to assess recoverability (i.e., ability of a product to recover affected data and re-establish the state of the system in the event of a failure).

In this dissertation, we develop these concepts in the context of Network Function Virtualization, by identifying possible use cases and context for a dependability benchmark, and by proposing appropriate measures and faults for this domain.

2.2 Fault Injection

2.2.1 Introduction

In the context of business- and mission-critical scenarios, intense testing activities are of paramount importance to guarantee that new systems and their built-in fault-tolerance mechanisms are behaving as expected. In such contexts, ensuring that the system behaves properly in the presence of a fault is a problem that requires something more than traditional testing. *Fault injection is the process of introducing faults in a system, with the goal of assessing the impact of faults on performance and on continuity of service, and the efficiency (i.e., coverage and latency) of its fault tolerance mechanisms.*

Fault Injection is a kind of what-if experimentation, and it may take place along with, or after, other testing activities. The target system is exercised with a workload, and faults are inserted into specific software components of the target system. The main goal is to observe how the system behaves in the presence of the injected faults, by reproducing component faults that will eventually affect the system during operation. Fault injection is used in several scenarios: to validate the effectiveness and to quantify the coverage of software fault tolerance, to assess risk, and to perform dependability benchmarking [159, 30, 82].

2.2.2 Overview of Fault Injection Testing

The development of fault injection approaches followed the evolution of computer systems. In the beginning, only simple hardware systems were used in the most critical application sectors. Thus, first fault injection approaches consisted of injecting physical faults into the target system hardware (e.g., using radiation, pin-level, power supply disturbances, etc).

The growing complexity of the hardware turned the use of these physical approaches quite difficult or even impossible, and a new family of fault injection approaches based on the runtime emulation of hardware faults through software (Software Implemented Fault Injection - SWIFI) become quite popular. SWIFI tools emulate the *effects* of hardware faults on software, thus avoiding the costs and the complexity of physical fault injection. SWIFI injects corruptions in the program state (e.g., data and address registers, stack and heap memory) and in the program code (e.g., in memory areas where code is stored, before or during program execution), using simple hardware fault models, such as bit flipping or bit stuck-at.

With the spread of computer systems in many application domains, such as transportation, telecommunications, and e-commerce, we are witnessing an increasing complexity of the software part of these systems, which became a non-negligible cause of IT failures. The recent outages of leading IT services providers, such as Amazon, Google and Microsoft [48, 135, 100], were in fact due to software bugs, incorrect administration actions, and excessive load. Thus, more recent fault injection approaches, which are referred to as Software Fault Injection (SFI), were developed to extend the scope of fault injection to faults related to software.

In practice, Software Fault Injection consists of the introduction of corrupted inputs and outputs values at the interface of software components (such as at the Application Programming Interface of the component), and of small changes in the code of the component. The use of SFI has been recently recommended by several safety standards, such as the ISO 26262 standard for automotive safety [75], which prescribes the corruption of software components for evaluating error detection and handling mechanisms in software, and the NASA standard 8719.13B for software safety [107], which recommends fault injection to assess system behavior in the presence of faulty off-the-shelf software.

2.2.3 Basic Concepts of Fault Injection Experiments

Throughout this dissertation, we will use the terminology defined by Avizienis et al. in [15]. A *fault* is the adjudged or hypothesized cause of an incorrect system state, which is referred to as *error*. A *failure* is an event that occurs when an incorrect service is delivered, that is, an error state is perceived by users or external systems.

Fault injection experiments follow the common schema presented in [71], and shown in Figure 2.2. The system under analysis is usually named *target*. There are two entities that stimulate the system, respectively the *load generator* and the *injector*. The former exercises the target with inputs that will be processed during a fault injection experiment, whereas the latter introduces a fault in the system. The set of inputs and faults submitted to the system are respectively referred to as *workload* and *faultload*, which are typically specified by the tester through a *library* by enumerating inputs/faults or by specifying the rules for generating them. A fault is injected by tampering with the state of the system or with the environment in which it executes. Fault injection usually involves the execution of several *experiments* or *runs*, which form a *fault injection campaign*, and only one or few faults from the faultload are injected during each experiment.

The *monitor* entity collects from the target raw data (*readouts* or *mea-*

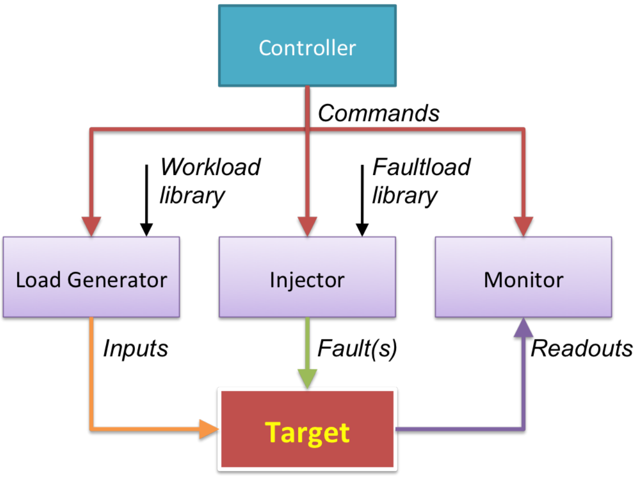


Figure 2.2. Conceptual schema of fault injection [71].

surements) that are needed to evaluate the effects of injected faults. The choice of readouts depends on the kind of system considered and on the properties that have to be evaluated. They may include the outputs of the target (e.g., messages sent to users or to other systems) and the internal state of the target or its parts (e.g., the contents of a specific area of memory). Readouts are used to assess the outcome of the experiment: for instance, the tester can infer whether the injected fault has been tolerated, or the system has failed. In order to obtain information about the outcome of an experiment, readouts are usually compared to the readouts obtained from fault-free experiments (referred to as *golden runs* or *fault-free runs*). All the described entities are orchestrated by the *controller*, which is also responsible for iterating fault injection experiments forming the fault injection campaign as well as for storing the results of each experiment to be used for subsequent analysis.

Let's now see the phases of a fault injection experiment. Initially the system is assumed to work in the "Correct" state. As soon as a fault is injected and a workload is applied, two behaviors can be observed. First, the fault is not activated and it remains latent. In this case, after a timeout the experiment runs out and no failure is produced. Second, the fault is activated and it becomes an error. At this stage, an error i) may propagate, by corrupting other parts of the system state until the system exhibits a failure, ii) can be latent in the system, and iii) can be masked by fault tolerance mechanisms. On the basis on the collected readouts, the monitor should be able to identify the previous effects as well as the case in which the fault is not activated at all.

In some cases, only a fraction of fault injection experiments are able to activate faults and to produce errors; the others are useless since no effects can be observed. In order to accelerate the occurrence of failures, a different, and cheaper, form of fault injection can be adopted, namely *error injection*. Here, the *effects* of faults are introduced in place of the actual faults. A well known technique of error injection is *Software-Implemented Fault Injection* (SWIFI), in which the effect of hardware faults (e.g., CPU or memory faults) are emulated by corrupting the state of the software, instead of physically tampering with hardware devices.

A very important aspect of fault injection is the set of *measures* that are adopted to evaluate the target system in the presence of faults. The goal of these measures is to represent the *ability to tolerate faults* of one or more *fault tolerance algorithms and mechanisms* (*FTAMs*) in the system. The effectiveness of fault tolerance can be hampered by flaws in the design or implementation of FTAMs, which cause the lack of *error and fault handling coverage*. Moreover, fault tolerance can be affected by the lack of *fault assumption coverage*, that is, developers can make incorrect or incomplete assumptions about faults that can occur during operation (e.g., assumptions about how a faulty component would behave, or on the independence between multiple faults) [14, 127].

Given an FTAM, its effectiveness is quantified using a *coverage factor*

[14, 127], which is defined as the *conditional probability* that a fault is correctly handled, given the occurrence of a fault and an input workload. In practice, the coverage factor of an FTAM can be evaluated by measuring the percentage of fault injections that are tolerated, and the number of failures caused by fault injections. It is important to note that the fault tolerance of a system is strongly dependent on the faults and inputs that are faced during operation: thus, fault injection experiments should realistically emulate these faults and inputs. Another measure of fault tolerance effectiveness is the *mean coverage time* (also referred to as *latency*) [14], that is, the expected time required to handle a fault, which can also be estimated using fault injection.

2.2.4 Key Properties of Fault Injection Testing

The quality of results obtained by Fault Injection is strongly dependent on several key properties of the experiments, namely:

- **Representativeness** refers to the ability of the faultload and the workload to represent the real faults and inputs that the system will experience during operation. If this is not the case, fault injection tests will be ineffective and not meaningful. Representativeness of faultloads is achieved by defining a realistic fault model, and by using a tool that accurately reproduces this fault model during an experiment.

- **Usability** requires to support developers in applying fault injection quickly and with low effort, by making *easy the setup and execution* of fault injection experiments, and the *analysis* of results. It is desirable that Fault injection tools are *reusable* across different target systems, and can support several *fault models*.

- **Reproducibility** is the ability of obtaining the same (or statistically equivalent) results when fault injection experiments are repeated, in order to allow developers to reproduce a failure discovered by fault injection, and to fix the system.

- **Non-intrusiveness** of fault injection tools is required in order to produce accurate measures. To be non-intrusive, the presence of the tool in the target system (e.g., for inserting faults and for collecting data) should not introduce significant perturbations that would distort the results of the experiments.

- **Efficiency** is the ability to achieve relevant and useful results with a reasonable number of experiments. A fault injection approach is not efficient when it requires an unfeasibly high number of experiments to obtain a statistically significant validation of the system, and to discover FTAM issues. To achieve efficiency, a fault injector should be designed to inject faults that actually put the target system under stress (i.e., to avoid the injection of faults that cannot have an impact on the target, such as faults in unused components and interfaces) and that provide useful feedback (i.e., to avoid the injection of redundant faults that would provide identical results).

Virtualization is an enabling technology to set up a cloud computing infrastructure. Virtualization allows to abstract physical resources (e.g., CPUs, network devices, storage devices, and so on) in order to share and to provide resources, making a physical machine as a soft component to use and manage very easily. Virtualization software is used to run one or more so-called *Virtual Machines* (VMs) (a software abstraction of a physical machine) on a single physical machine, providing the same functionalities as if they were many physical machines. This virtualization software is named *Hypervisor*, which is responsible of executing and managing multiple VMs in order to synchronize the access to the CPU, memory and other I/O resources of the physical machine.

Therefore, to assure the reliability of cloud systems, it is necessary to assess the reliability of the virtualization environment as a whole, focusing both on VMs and on the Hypervisor, as well as on the Cloud Management Stack software that orchestrates them (such as the well-known OpenStack framework) to efficiently manage cloud infrastructures. In the subsequent sections, we present an overview of related studies that adopt fault injection to assure a high-level of reliability of cloud systems, focusing on Virtual Machines (subsection 2.3), Cloud Management Stack (subsection 2.4), and Hypervisors (subsection 2.5).

2.3 Fault Injection Testing of Virtual Machines

Testing in distributed and parallel systems in many cases is an hard task due to complexity of such systems in operational phase. In particular, cloud-based systems have to be high dependable, thus they must have fault tolerance mechanisms that consider not only software failures but hardware failures too. In this section, we consider fault injection testing of Virtual Machines, focusing on user software and operating system (e.g., Linux) that runs on those VMs. This fault injection testing aims to highlight reliability issues against hardware faults. The latter are emulated within VM, just modifying virtual devices, such as hard disk and network controller, and tampering memory and CPU state. Moreover, that approach includes software bug at operating system's level, by corruption of source code into memory.

In this section, we present D-Cloud and DS-Bench Toolset, fault injection tools that adopt virtualization for performing Fault Injection Testing of distributed software.

2.3.1 D-Cloud and DS-Bench Toolset

D-Cloud [16] is a dedicated simulated test environment, based on QEMU for virtualizing physical machines, and on Eucalyptus cloud computing system for managing these VMs for testing purposes. D-Cloud adopts QEMU to emulate hardware faults, by injecting various typical faults into the guest OS.

Conversely, DS-Bench Toolset [59] is a framework that computes dependability metrics of the overall system under test (SUT), using various benchmark programs, by injecting anomaly loads; furthermore, it provides the evidence for the assurance case based on the benchmark results.

Faultload. D-Cloud considers hardware faults in memory, hard-disk and network devices. It performs fault injection by simulating data corruptions in the emulated devices. The fault types (see Table 2.1) include the corruption of individual sectors of the disk (e.g., the sector was damaged by wear-out), of packets sent through the network (e.g., loss or bit corruption of a packet), and of memory cells. Moreover, it can simulate a unresponsive or slow hard disk and network devices.

Table 2.1. D-Cloud Fault Types [16]

Physical Device	**Fault Type**
Hard Disk	Error of specified sector Specified sector is read-only Error detection by ECC Received data contains error Response of disk becomes slow
Network Controller	1bit error of packet 2bit error of packet Error detection by CRC Packet Loss NIC is not responding
Memory	Bit error Byte at specified address contains error

Architecture. In Figure 2.3 is shown D-Cloud architecture. There are

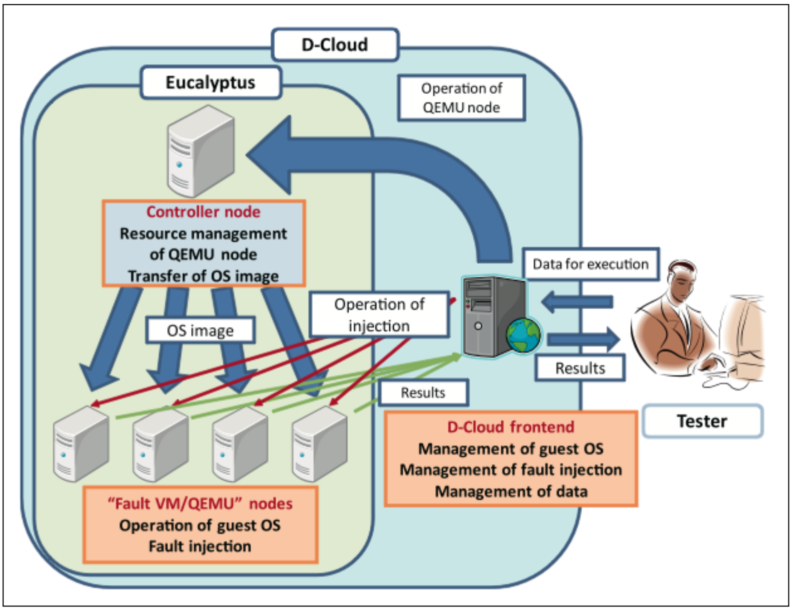


Figure 2.3. D-Cloud architecture [16]

three main components:

1. **QEMU nodes**: QEMU provides fault injection facilities leveraging on mechanisms that emulate many hardware devices. Moreover, being open source, allow to edit hardware emulation code in order to add fault injection code;

2. **Controller node**: it manages QEMU nodes exploiting Eucalyptus, which manages machine resources using virtual machines;

3. **D-Cloud front-end**: it provides an interface to the tester, that allows him to configure system test environments and logging mechanisms, and to analyze results.

A tester can describe a *test scenario* through an XML-based description format. A test scenario includes:

- the configuration of hardware environment, that specifies hardware configuration of the virtual machines (i.e., number of CPUs, number of NICs, amount of memory and ID of OS image to be booted);

- the configuration of software environment, that specifies a *system* definition, consisting of a number of *hosts* with specific configuration, that describe the installed applications and the network configuration;

- the fault injection definition, that specifies the target device, the fault type and the time duration of injection experiment;

- the configuration of test execution, that specifies the number of tests to be executed (*runs*), each of which include the finish time, and tester defined script to be executed in the test.

The D-Cloud front-end issues commands to the Controller Node (the Eucalyptus controller), which dispatches fault injection experiments to the available QEMU nodes. VMs are used to execute the software, to perform fault injection, and to take a snapshot of the system at the end of an experiment. Once tests are completed, the tester can download test outputs, system logs, and VM snapshots, analyze them and obtain results.

Figure 2.4 shows an overview of DS-Bench Toolset architecture, and consists of:

- **DS-Bench**, a benchmark test framework for system dependability. It includes a *controller*, that has a benchmark database, in which is stored benchmark scenario (described via XML), benchmark programs (e.g., *httperf*), anomaly generators and benchmark results;

- **D-Case Editor**, a requirements description and agreement support tool;

- **D-Cloud**, an execution testing environments (see above);

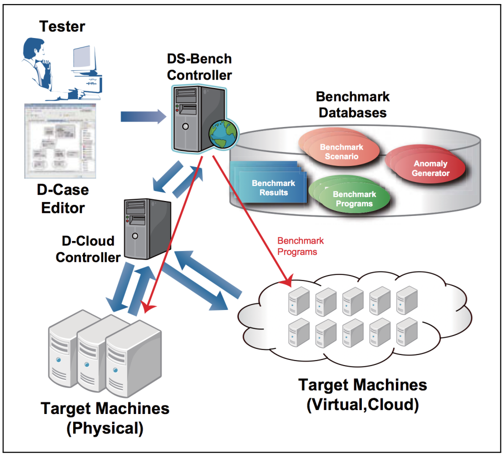


Figure 2.4. DS-Bench Toolset architecture [59]

In depth, a tester interacts with the toolset via the D-Case Editor (implemented using Eclipse Graphical Modeling Framework), that allows to specify a so-called D-Case *diagram*; that diagram contains some requirements, such as performance requirements (e.g., latency), or availability requirements (e.g., downtime less than a specific time).

D-Case Editor imports the suitable benchmark from benchmark database (if there is not exist, it creates new one), and then requests to DS-Bench to execute such a benchmark test. The target machines can be both physical and virtual machines. DS-Bench execute selected benchmark on target machine with the aid of D-Cloud controller.

D-Cloud controller generates anomaly loads which encompasses two main types: first, it include programs that run on target machines, which consume computing resources, such as CPU, memory and I/O bandwidth; second, injects fault from outside of target machines, both on physical machine (e.g., cut an external power source, or cut a physical link) and on virtual machine, by emulating hardware faults via VM injection (e.g., memory bit flip, disk I/O error, packet drop and so on). D-Cloud controller sends a list of all available resources to the DS-Bench controller, and then assigns target machines to execute the benchmark.

A similar study was presented [16], with the difference that in this work is used OpenStack, a popular open source cloud-management software that allow to manage and assign physical and virtual machine. Upon the test complete, D-Case Editor analyses results and checks if requirements are met.

Results. In this section, the results are related to the evaluation of the DS-Bench Toolset. The target is a system that simulates a typical web server system that hosts the MoinMoin Wiki system [105], simply a wiki engine that runs a wiki server. Figure 2.5 shows the target system, which consists of a front-end web servers with an Apache HTTP Server, that are set up as physical machines (server performance is much more critical than clients machine, so it is better test on the actual hardware), and the client that access the server, that are set up as virtual machines. The experiments are conducted focusing on web server failure, simulated via a script that cuts down a network link by controlling a network switch connected to the target physical machine, using the SNMP (Simple Network Management Protocol) protocol. As benchmark program is used a tool based on *httperf*. The benchmark takes 60 seconds and at $t = 30s$ is injected a network link failure. The requirement goal is to achieve a response time (latency) of web server within 3000ms, when the access rate is within 4 request/s, even in presence of network failure.

2.4 Fault Injection Testing of Cloud Management Stack

By Cloud Computing stack it is meant the set of software and of technologies responsible for the creation and management of cloud platforms, via the cooperation of distributed services. In particular these services take

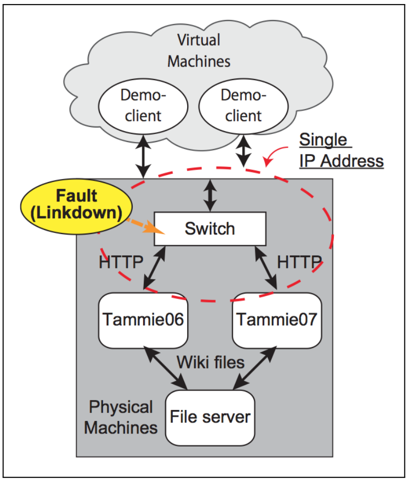


Figure 2.5. Target system of DS-Bench Toolset evaluation [59]

care about instantiation of VMs on hypervisor, VMs migration, creating and deleting VMs, storage facilities and network communication. Moreover, due to unreliable hardware platforms, cloud computing stack is also responsible for providing a protocol that recovers from various hardware failure such as machine crashes, disk errors and network errors.

In this section we overview two main studies, [79] is one of the firsts on fault resilience in OpenStack [120], a popular cloud management stack software, and [77], a *summa* of studies on multiple failure injection, which evaluate various cloud management software like HDFS [136], Cassandra [88] and Zookeeper [74].

2.4.1 OpenStack Fault Injection Testing

An important dependability aspect of cloud computing software is related to resiliency, still not studied in depth. For example, as soon as faults occur, VM creation or VM migration may fail or take much more time than expected, and VMs may be labelled as successfully created but probably with a lack of critical resources (e.g., network related, such as IP addresses). So, fault resilience issues may be lead to unusable and unstable cloud platform.

One of the most important open cloud computing software is OpenStack [120], that controls compute, storage and networking resources in a whole data center, managed and provisioned through a web-based dashboard, command-line tools or a RESTful API. The goal of OpenStack is to allow the creation of private or public cloud computing platforms, simple to implement and massively scalable.

Ju et al. [79] presents a systematic study on fault resilience of this cloud computing software. The proposed framework injects network faults targeting communications among OpenStack's services like compute, image and identity services, but also database, hypervisor and messaging services. The authors evaluate the framework on two OpenStack versions, identifying bugs, such as timeout between services communication, or lack in periodic checking of service liveness (VM creation API has completed its job?) and so on, more described in the next sections.

Faultload. The fault injection is focused on the **execution graph**, that is the execution of OpenStack during the processing of an external request. OpenStack uses two main communications mechanisms: compute services (see below) use *Remote Procedure Calls* (RPCs) for internal communications within the *service group*; the other services, conform to *REpresentational State Transfer* (REST) architecture, communicate with each other through the *Web Server Gateway Interface* (WSGI) (for more details on OpenStack architecture see [121]).

The execution graph is generated starting from logs of fault-free request processing task, where:

- each vertex represents a *communication event*; each communication event is a pair of the communication entity(e.g., *image-api* service) and the *communication type* (e.g., REST request and send opera-

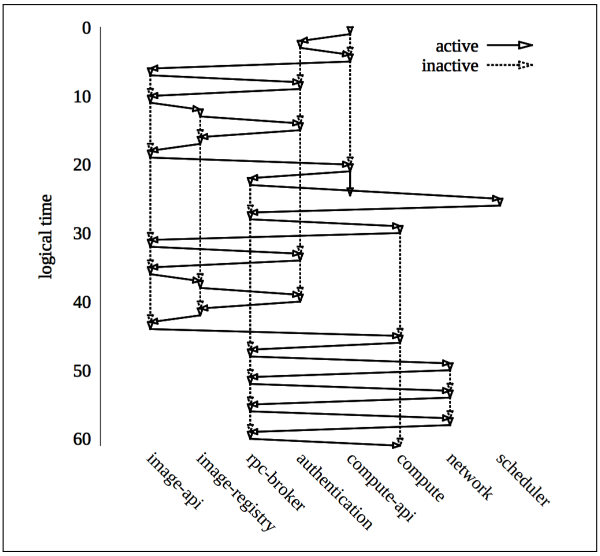


Figure 2.6. Example of OpenStack execution graph

tion);

- each edge describes the causality among events.

Figure 2.6 shows an example of execution graph related to VM creation, consisting of REST and RPC communications.

The external APIs are related to the following services:

- Compute service, that allow to manage VMs (e.g., creating and deleting VMs);
- Image service, that allow to manage VM images (e.g., registration and deployments of VMs);
- Identity service, that allow to manage users, tenant and roles, and authentication related issues.

The authors consider only two common faults, well studied in literature:

- Server (or service) crash, lead to the unavailability of specific service. This fault is injected simply by killing important service processes using *systemd*, a Linux service manager;
- Network partition, lead to network split, that is no communication between two subnets. This fault is injected installing *iptable* rules on service hosts that should be partitioned.

Architecture. Basically, this approach performs fault injection considering the evolution of the request execution, and the state of the overall system, focusing on state transitions. Figure 2.7 shows the framework, that consists of three main components:

1. **Logging and Coordination module**: this module is responsible for log OpenStack services communications and for coordinating the Fault-Injection module with the execution of OpenStack;
2. **Fault-Injection module**: this module is composed by a *fault-injection controller*, that runs on a test server node, and a *fault-injection stub*, that runs with OpenStack. Essentially, using information gathered by logging and coordination module, the fault injection controller decides when to inject faults, and triggers the fault-injection stub component;
3. **Specification-Checking module**: this module verifies whether, after each fault-injection experiment, the behavior of OpenStack is compliant to a predefined behavioral model (i.e., whether OpenStack is able to provide a reliable behavior).

After logging OpenStack communications in a fault-free execution, the Fault-Injection module parses logs and creates a fault-free execution graphs; according to the latter, and with a predefined fault specification, the Fault-injection module generates a set of test plans. These test plans

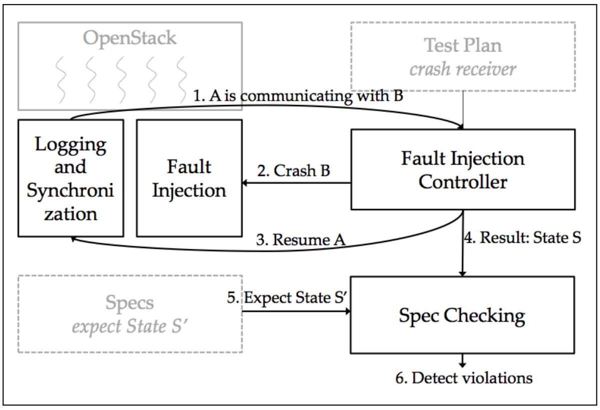


Figure 2.7. Fault Injection framework

are generated exhaustively, considering all the fault types applicable, with any specific criteria (e.g., cluster execution graph vertices within a testing priority). Each test plan consists of an execution graph, a fault type (specified in fault specification) and a fault location (computed through fault-free execution graph). At this point, Fault-injection controller actual makes fault-injection decision and communicates it to the stub that actual injects faults, according to the test plan. Upon test completion, Fault-injection module collects results and sends them to the Specification-Checking module, which checks results against specification, and it reports possible violations.

Since OpenStack developers publish only a few state-diagram transitions, the authors manually generate specification by taking into account OpenStack developers' assumption and overall behavior of the system. Database-related specifications are written through *SQLAlchemy* library [140], and for others a generic Python scripts are used.

Regarding the VMs *state* in OpenStack, Algorithm 1 shows a specification example of *VM State Stabilization*: this specification checks if a VM, after creation, reaches an ACTIVE state (i.e., stable state) instead of remaining in a BUILD state (i.e., transient state). Algorithms 2 and 3 shows others examples of specifications, respectively *Ethernet Configuration*, that checks if an ACTIVE VM has the ethernet controller properly configured, and *Image Local Store*, that checks if the database of OpenStack image service is synchronized with the view of filesystem on the service host.

Algorithm 1 VM State Stabilization Specification

```
query = select VM from compute_database where VM.state in collection(VMunstable states)
if query.count() = 0 then
    return Pass
end if
return Fail
```

Algorithm 2 Ethernet Configuration Specification

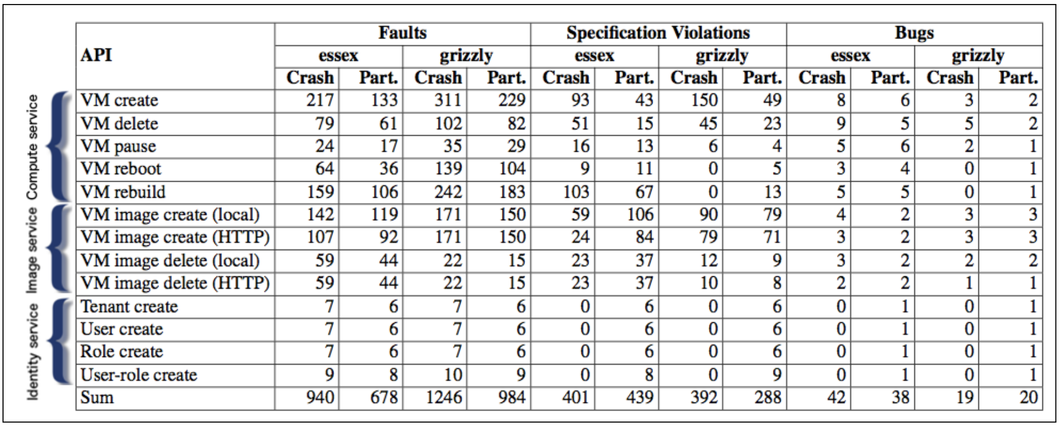
```
if VM.state() = ACTIVE and ( (VM.host.Ethernet not setup) or (network_controller.Ethernet not setup) ) then
    return Failt
end if
return Pass
```

Algorithm 3 Image Local Store Specification

```
query = select image from image_database where image.location is local)
if local_image_store_images = query.all() then
    return Pass
end if
return Fail
```

Results. In [79], testing was performed on two OpenStack versions, *essex* and *grizzly*. By injecting crash and partition faults, Table 2.2 shows the number of injected faults, the number of specification violation and the number of discovered OpenStack software defects (bugs). As mentioned

Table 2.2. OpenStack Fault Injection results [79]



	API	Faults				Specification Violations				Bugs			
		essex		grizzly		essex		grizzly		essex		grizzly	
		Crash	Part.	Crash	Part.	Crash	Part.	Crash	Part.	Crash	Part.	Crash	Part.
Compute service	VM create	217	133	311	229	93	43	150	49	8	6	3	2
	VM delete	79	61	102	82	51	15	45	23	9	5	5	2
	VM pause	24	17	35	29	16	13	6	4	5	6	2	1
	VM reboot	64	36	139	104	9	11	0	5	3	4	0	1
	VM rebuild	159	106	242	183	103	67	0	13	5	5	0	1
Image service	VM image create (local)	142	119	171	150	59	106	90	79	4	2	3	3
	VM image create (HTTP)	107	92	171	150	24	84	79	71	3	2	3	3
	VM image delete (local)	59	44	22	15	23	37	12	9	3	2	2	2
	VM image delete (HTTP)	59	44	22	15	23	37	10	8	2	2	1	1
Identity service	Tenant create	7	6	7	6	0	6	0	6	0	1	0	1
	User create	7	6	7	6	0	6	0	6	0	1	0	1
	Role create	7	6	7	6	0	6	0	6	0	1	0	1
	User-role create	9	8	10	9	0	8	0	9	0	1	0	1
	Sum	940	678	1246	984	401	439	392	288	42	38	19	20

in 2.4.1, the tests are performed on 11 different external OpenStack APIs. The authors have injected 3848 faults, implemented 26 specifications to check the overall OpenStack behavior, detecting 1520 violations and finally, using the proposed framework, they have identified 23 bugs categorized in the Table 2.3.

Table 2.3 shows the number of discovered bugs, which are classified using seven categories:

- **Timeout bugs** are related to a service that is in an indefinite waiting (or blocked) for a service response. For example, in OpenStack *essex* version, timeout mechanisms, in REST communications, are not properly implemented. Thus, a service may be blocked for a response from another service, via the REST mechanism, if the two services become network-partitioned (for example, after the service request is sent but before the response from the other service is received);

- **Periodic checking bugs** are related to the lack of checking mechanisms, that are in charge of monitoring service liveness, resume interrupted execution, clean up garbage and prevent resource loss. For example, in the VM creation request, VM state transits from *NONE* (i.e., not exists) to BUILD (i.e., under construction) to ACTIVE (i.e., actively runs). If the creation process is interrupted just after reaching the BUILD state, it is possible that VM indefinitely remains in that state.

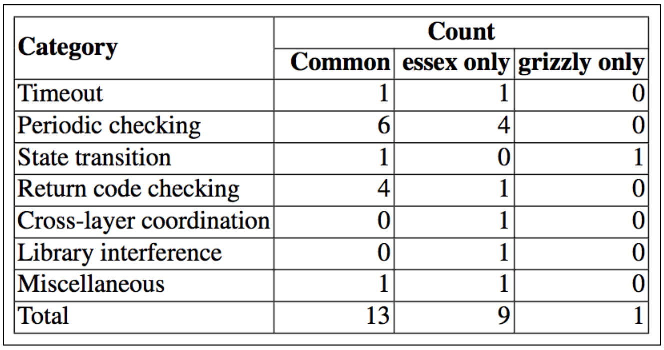
- **State transition bugs** are related to the state-transition diagrams held in OpenStack database. For example, the OpenStack *grizzly* version periodically checks if a VM exceeds the maximum given time allowed to remain in BUILD state; if so, OpenStack convert a VM from BUILD to ERROR state, without cancel the VM creation request related to that VM. Thus, if a transient fault occurs during VM creation request, then a VM may transit from BUILD to ERROR and then to ACTIVE, because the VM can be created, after periodical check described above, and after the faulty conditions are not present anymore.

- **Return code checking bugs** are related to erroneous return code of OpenStack services. For example, during the VM creation process, if the OpenStack *identity service* can not authenticate a user token passed from a compute API service, then returns an error code to that service. Due to implementation error within the checking logic, the compute API service considers such an error due to a wrong token, producing an improper error message.

- **Cross-layer coordination bugs** are related to incorrect interpretations of behaviors of different layers in the OpenStack architecture (see [121]).

- **Library interference bugs** are related to using of various external libraries that may lead to unexpected behavior. For example,

Table 2.3. OpenStack bug categories [79]



Category	Count		
	Common	essex only	grizzly only
Timeout	1	1	0
Periodic checking	6	4	0
State transition	1	0	1
Return code checking	4	1	0
Cross-layer coordination	0	1	0
Library interference	0	1	0
Miscellaneous	1	1	0
Total	13	9	1

OpenStack uses patched version of Python library functions to support thread scheduling; incompatibility in the patched functions may highlights bugs hard to detect.

- **Miscellaneous bugs** are related to implementation bugs which not fall into above categories. For example, in the OpenStack *essex* version, if a connection opening procedure is interrupted, a subsequent *open* call is issued without first invoking *close* to clean up the state of the connection. When retrying to *open* connection all attempts fail with an "already open" error message.

2.4.2 PreFail

A recent trend on fault injection research is to consider the injection of multiple faults during the same experiment. Multiple injections are motivated by the observation that, even if multiple faults are rarer than single ones, they tend to be neglected during development and testing, and are often less tolerated by a system. Moreover, with the growing scale of computer systems (see cloud computing systems) and the higher degree of complexity and integration of hardware and software components, multiple faults are regarded as a likely future trend.

PreFail [77] allows to deal with a very high number of injection experiments, that arises from the "combinatorial explosion" of multiple injections. The PreFail tool allows the tester to control fault injection using *pruning policies*, which select the combinations of faults to be injected during experiments. The policies offered by PreFail are oriented towards selecting a small set of faults, and to maximize the *efficiency* of fault injection tests. This goal is reached by letting the user to specify a pruning policy. Examples of policies are:

- **Failing component**: Let's assume a scenario in which a tester wants to test a four-nodes system (each of which holds a replica of the same file), focusing on the used distributed write protocol. Instead of injecting all possible combination of node failures, a tester can specify a policy that injects only in a subset of nodes.

- **Failure type**: Let's assume a scenario in which the tester knows that the target is a *crash-only* system (i.e., a system that handles any I/O failure simply by translating them into system crash, and then restarts the system as a recovery action). The tester may want to inject only I/O failures, without considering crash failures, because any failure lead to a crash anyway;

- **Domain-specific optimization**: Let's assume a scenario where the OS buffers a file after the first read (which usually happens in commodity OSs). Considering ten consecutive reads from the same input file, a tester could run ten experiments, in which a disk failure is injected at each call. But, bearing in mind the OS buffering, it is unlikely to have a failure after the first read operation. Thus, injecting only at the first read operation may reduce the space of multiple failures;

- **Failing probabilistically**: a tester may want to inject only multiple failures that have an occurrence likelihood greater than a specific threshold. Furthermore, based on statistical data, a tester can implement policies that leverage on some failure probability distribution.

Finally, PreFail also supports **coverage-based** policies, that are related to:

- **Code-Coverage**, which aims to achieve a high source code coverage, with the minimal number of experiments. An example is to achieve a high coverage of source location of I/O calls, very failure-prone;

- **Recovery-Coverage**, which aims to cover recovery paths in the code, such as the code of recovery procedures and exception handlers.

Thanks to small and highly-expressive programs, policies can be customized to account for domain knowledge, for instance by focusing the type and location of faults that developers know to be most likely, and by considering the allocation of processes across nodes and racks to inject more realistic faults (e.g., network issues among processes that are actually remote).

PreFail takes into account various faults, such as hardware fault related to I/O (e.g., network and hard disk) and software faults (e.g., process crash).

Finally, the considered system targets are three important cloud software systems:

- Hadoop Distributed FileSystem [136], a storage system for Hadoop MapReduce [68];

- Cassandra [88], a distributed storage system;

- Zookeeper [74], a service for coordinating processes of a distributed system.

Faultload. In PreFail, the injection approach is to instrument the target with the aim of raising an exception (the specific type depends on the particular operation being performed) instead of normally executing a system call. Furthermore, that exception must be handled by a specific recovery protocol.

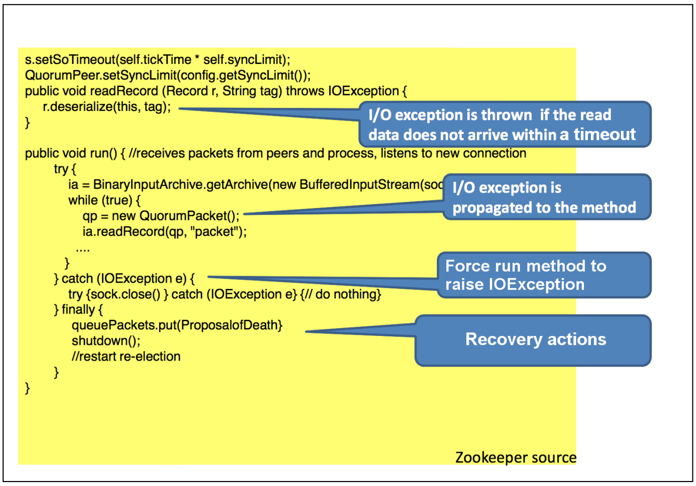
- **Network fault**: such type of faults, for example, are related to network partition problems. These can be triggered by an anomaly in network (e.g., link failure, loss packets), in a node (process crash, CPU overloading, unhandled exceptions) or disk (sectors corrupted, increase latency). For example, in order to fail a network connection between two nodes, or to isolate a node among others, a tester could inject a fault forcing the system call, that performs network I/O, to throw an exception (e.g., *NoRouteToHostException*, *PortUnreachableException* and so on) and to return an error;

- **Hard Disk fault**: such a type of faults are related to all the hard disk and data corruption problems. As for network faults described above, to trigger a disk failure, a tester can force a system call, that performs I/O operations (read, write, append), to throw an I/O exception and return unsuccessfully. Instead, to point out a disk data corruption, it can force the read system call to return random data;

- **Software fault**: such type of faults are software bugs. For example, a tester can force to kill a process, which may involve in other operations that can slow down the overall system, or crash the system.

Figure 2.8 shows a code snippet example from Zookeeper application, that provide consistence and partition-tolerant service. A network failure can be triggered by force the *run* method to throw an *IOException*.

Architecture. In order to inject different type of failures, the authors introduce the *failure-injection task* (FIT), a pair of a *failure-injection point*



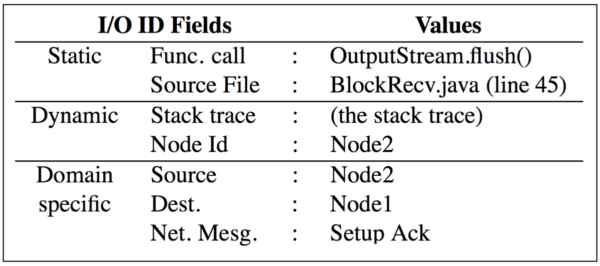
```
s.setSoTimeout(self.tickTime * self.syncLimit);
QuorumPeer.setSyncLimit(config.getSyncLimit());
public void readRecord (Record r, String tag) throws IOException {
    r.deserialize(this, tag);
}

public void run() { //receives packets from peers and process, listens to new connection
       try {
          ia = BinaryInputArchive.getArchive(new BufferedInputStream(soc
          while (true) {
             qp = new QuorumPacket();
             ia.readRecord(qp, "packet");
            ....
          }
       } catch (IOException e) {
          try {sock.close() } catch (IOException e} {// do nothing}
       } finally {
           queuePackets.put(ProposalofDeath}
           shutdown();
           //restart re-election
       }
}
```

I/O exception is thrown if the read data does not arrive within a timeout
I/O exception is propagated to the method
Force run method to raise IOException
Recovery actions
Zookeeper source

Figure 2.8. Code snippet example of Zookeeper application

(FIP), that abstract an I/O call (e.g., system/library calls that perform disk or network I/Os) in which a failure can be injected, and a failure type (e.g., crash, disk failure and so on). As Figure 2.9 shows, a FIP can be generated both from static (e.g., system call, source file) and dynamic information (e.g., stack trace, node ID) available at that point, and from domain-specific information (e.g., in network domain, source and destination node ID, or network exchanged messages). Furthermore, Figure 2.9 shows information fields described above and related FIP values: for example, a FIP related to a function call has function call name as value. Finally, a FIT allows to trace the sequences of failures that have been explored, and sequences not explored yet.

Figure 2.10 presents an overview of the architecture of testing framework. *Workload driver* is the component in which a tester can specify workloads related to targets (e.g., HDFS write, read, append etc.), and the maximum number of failures injected per run. The *Target System* is instrumented by adding a *Failure Surface*, that builds a FIP at each I/O point (an execution of an I/O call) and checks wether a previously injected



I/O ID Fields			Values
Static	Func. call	:	OutputStream.flush()
	Source File	:	BlockRecv.java (line 45)
Dynamic	Stack trace	:	(the stack trace)
	Node Id	:	Node2
Domain specific	Source	:	Node2
	Dest.	:	Node1
	Net. Mesg.	:	Setup Ack

Figure 2.9. Example of FIP [67]

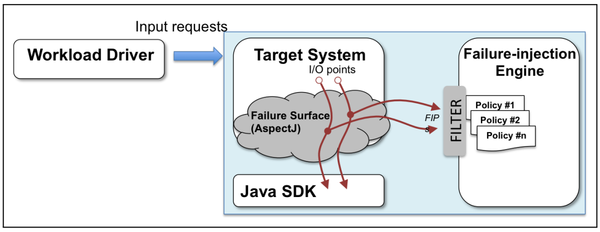


Figure 2.10. PreFail architecture [67]

failure affects the I/O point (this is the case in which we have a permanent failures); if so the Failure Surface returns an error code to emulate the failure. Otherwise, when the workload runs, Failure surface sends the generated FIPs to the *Failure Server*, which combines the latter with all potential injectable failures into the FITs. The latter makes a failure decision that satisfies tester defined policies, and sends it back to the failure surface. The workload driver stops when the failure server does not have any failure injection scenarios to run.

In more detail, the algorithm of PreFail's test workflow, outlined in Figure 2.10, takes the maximum number of of failures to inject in an execution of the system target and it iterates over N starting from 0. At step i, the Failure-injection engine executes the target system for each different *failure sequence* (i.e., a sequence of FITs) of length i. For i equal to zero, the Failure-injection engine executes the target without any injected fail-

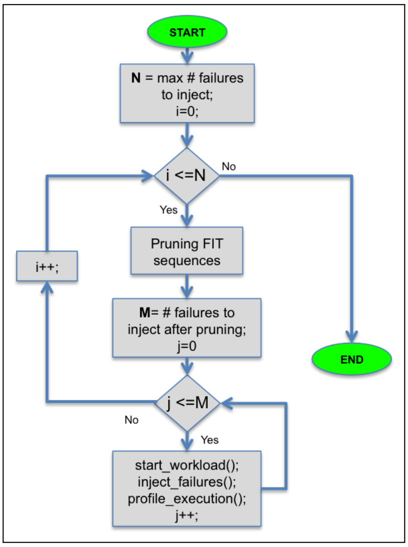


Figure 2.11. PreFail Test workflow

ure, it observes the FIPs that are seen during execution, computes FITs from them, and adds singleton failure sequences with these FITs, which Failure-injection engine can exercise in the next step (i.e. for $i = 1$). Furthermore, for i from 1 to N, PreFail prunes down the failure sequence, according to tester specified policies; then, for each failures, Prefail executes the target with the specified workload, injects the failure sequence and profile the target execution, observing new FIPs and creating a new FITs of length $i{+}1$ added to the next failure sequence to be exercised.

In [77] the Failure Surface interposes Java calls, which emulate hardware failures, to all I/O calls, supporting different failure types, such as disk failure, network partitioning (at node and rack levels) and software crash, by instrumenting target system via AspectJ [56], an aspect-oriented extension for Java language. Figure 2.12 shows the *aspects* used to intercept when a node becomes the leader or release its leadership in Zookeeper. When that node become leader, it starts executing the *lead()* method in

```
before() : ( call (void Leader.lead()) ) {
 setAsZkLeader();
}
after() : ( call (void Leader.lead()) ) {
 removeAsZkLeader();
}
```

Figure 2.12. AspectJ Example

```
def flt (fs):
   last = FIP (fs [ len(fs) - 1 ])
   return not explored (last, 'loc')
```

Figure 2.13. Example of source location filter policy

Leader.java, and set that node as leader (*setAsZkLeader() call*); the node exits the *lead()* method then it is no longer the leader (thus, *setAsZkLeader()* is invoked).

PreFail considers two main types of policies: *filter policies*, that remove experiments that do not comply to a user-defined condition; and *cluster policies*, that remove experiments that are equivalent to other experiments according to a user-defined condition. Figure 2.13 and Figure 2.14 are examples of policies aimed at increasing code coverage (discussed above): the first policy filters experiments whose last fault injection point code location has already been covered by a previous experiment, and the second policy clusters experiments whose last FIPs have the same code location. Using libraries provided by PreFail (e.g., functions exposing fault injection points), policies are typically very small, with an average of 17 lines of Python code [78].

Results. The main target of PreFail evaluation is the HDFS used by Cloudera Inc. [33]. In [77] the recovery protocol of HDFS was evaluated against disk/network failures. Fault injection tests were performed under

```
def cls (fs1, fs2):
   last1 = FIP (fs1[len(fs1) - 1])
   last2 = FIP (fs2[len(fs2) - 1])
   return (last1['loc'] == last2['loc'])
```

Figure 2.14. Example of source location cluster policy

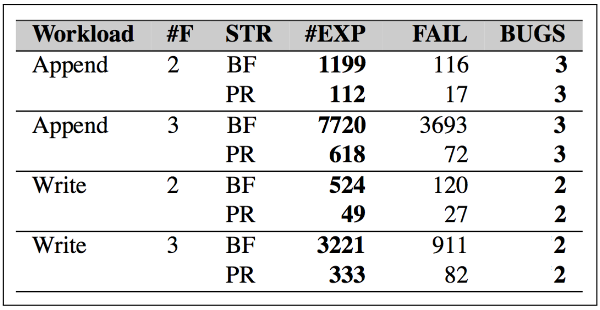
four different workloads: log recovery (e.g., recovery lost data), read, write and append (respectively, a high volume of read, write and append requests is submitted). PreFail found 6 known bugs (in version 0.20.0) and 6 unknown bugs (in version 0.20.2+737), which caused the loss and the unavaibility of files, and the unresponsiveness of HDFS components (e.g. master node).

The authors found an important bug related to recovery mechanisms in HDFS, and related to the append protocol. The latter is in charge of atomically appending new bytes to three replicas of a file, that are stored in three nodes. By injecting two node failures, PreFail found a recovery bug, in which the append protocol returns an error and the alive replicas are in an inaccessible state, since it is not clean.

Finally, the authors prove the efficiency of proposed filter mechanisms, that prunes down the number of executed experiments, thus the overall testing time. Figure 2.15 shows the results considering two different workloads, write and append, with two and three failures per run, and a policy that filter crash-only failures on disk I/Os in datanodes: the number of experiment, using the policy, is reduced by an order of magnitude, still finding the same number of bugs.

2.4.3 The Netflix Simian Army

In a cloud infrastructure, many factors like software bugs and hardware failures are beyond of our control, thus failures are not predictable and



Workload	#F	STR	#EXP	FAIL	BUGS
Append	2	BF	**1199**	116	**3**
		PR	**112**	17	**3**
Append	3	BF	**7720**	3693	**3**
		PR	**618**	72	**3**
Write	2	BF	**524**	120	**2**
		PR	**49**	27	**2**
Write	3	BF	**3221**	911	**2**
		PR	**333**	82	**2**

Figure 2.15. PreFail optimization results: *F* is the number of injected failures per run , *STR* is exploration strategy, *EXP* is the combinations/experiments, *FAIL* are the failed experiments, and *BUGS* are the bugs found. *BF* and *PR* stands for brute-force and prioritization (using the filter policies) respectively. [67]

building a reliable service is a hard challenge. An approach to build more reliable system is to deliberately introduce failures, in order to validate the system resiliency, that is, the ability of the system to provide services even in presence of a fault. Reducing the uncertainty related to faults allows to reduce the likelihood that the system behaves not properly. The idea behind the Netflix Simian Army is to deploy an application or an entire system over several VM instances, leveraging on Amazon Web Services (AWS). Then, it randomly injects CPU, disk, network, and other faults to check whether the system is still working properly.

In this section, we overview the Netflix's approach, *The Simian Army* [151], a set of tools (named *monkeys*) that allow to inject faults into a cloud computing platform, specifically built within AWS. *Simian Army*'s "monkeys" for assessing resiliency are:

- **Chaos Monkey**, that allows to randomly terminate virtual instances (i.e., virtual machines) in the production environment;

- **Chaos Gorilla**, that allows to cause an entire data center (e.g., an Amazon availability zone (AZ)) to go down;

- **Chaos Kong**, that allows to bring down an entire region, made up of multiple data centers;

- **Latency Monkey**, that allows to inject faults that simulate partially healthy instances.

Faultload. The Simian Army encompasses several fault types, tailored for each "monkey" described above. Currently, Netflix developed only the Chaos Monkey and their related fault types. Fault injection is performed by executing a script that simulates a specific type of fault. In the following, we show fault types of the Chaos Monkey tool, each of which is linked with a specific script that simulates them:

- **BurnCPU**: executes a CPU stress program on the node, using up all available CPU. This fault simulates either a *noisy neighbor* problem, that is, a VM that uses a huge amount of CPU resource resulting in a performance degradation for other VMs; this fault also simulates general issues with a CPU;

- **BurnIO**: executes a disk I/O stress program on the node, reducing the general I/O capacity. As before, this fault simulates either a *noisy neighbor* problem (in this case related to I/O resources), or just a general issue with the disk;

- **DetachVolumes**: force detach operation of all the EBS volumes. This fault simulates a catastrophic failure of an EBS, even though the instance could be still keep running (e.g., it should continue to respond to pings);

- **FailsDNS**: blocks TCP and UDP port 53 in order to simulate DNS resolution failures;

- **FailDinamoDB**: Adds entries to /etc/hosts so that DynamoDB API endpoints are unreachable.

- **FailEc2**: Adds entries to /etc/hosts so that EC2 API endpoints are unreachable;
- **FailS3**: Adds entries to /etc/hosts so that S3 API endpoints are unreachable;
- **FillDisk**: creates a huge file on the root device so that the disk fills up. This fault simulates a disk problem;
- **KillProcess**: kills processes on the node. This fault simulates the process crashing due to any reason;
- **NetworkCorruption**: induces network packet corruption using traffic-shaping. This fault simulates a problem with network interface controller;
- **NetworkLatency**: induces network latency using traffic-shaping. This fault simulates a VM that is *unhealthy*, thus simulates a service degradation;
- **NullRoute**: creates null routes. This fault simulate a node not able to communicate with other nodes.
- **ShutdownInstance**: shuts down the instance using the cloud instance-termination API. This fault simulates a VM crash.

Architecture. As mentioned above, The Simian Army is a set of tools (monkeys) that provide fault injection facilities at different levels in a cloud computing system. Figure 2.16 shows that the injection can be focused on a single virtual machine, a set of virtual machines (i.e., an entire data center or availability zone in the Amazon jargon), and a set of multiple data centers (i.e., a region).

The *Chaos Monkey* is a service which randomly terminates individual virtual instances during runtime, leveraging on the AWS APIs. The Chaos

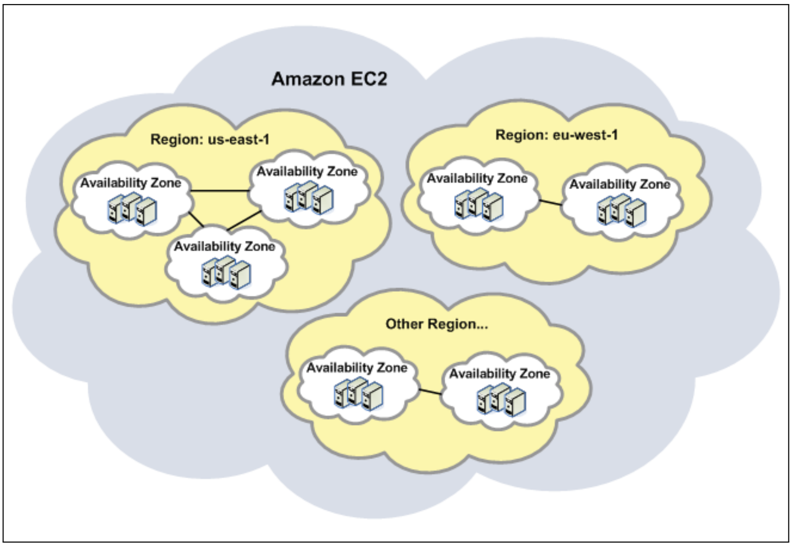


Figure 2.16. Amazon availability zones and regions

Monkey can be configured, for example, to wake up each hour and choose, with a specific probability, an instance to bring down. The failure of a virtual machine, for example, can be caused by emulating a power failure, a disk failure or a network failure (e.g., network partition). Through the Chaos Monkey, a system can be validated against single virtual machine failures. Instead, the *Chaos Gorilla* service assesses an entire data center, that consists of multiple virtual machines. It simulates a network partition failure, that is, the set of virtual machines, that belongs to a specific zone, can communicate each other but can not communicate outside the zone; furthermore, the Chaos Gorilla can simulate a *total zone failure* that consists in terminating all virtual machines in the zone. Finally, the *Chaos Kong* service allow to taking offline an entire region in order to prevent the isolation between such regions.

Fault injection is performed through the execution of a script on the node. For example, Figure 2.17, Figure 2.18 and Figure 2.19 show code

```
#!/bin/bash
# Script for BurnCpu Chaos Monkey

cat << EOF > /tmp/infiniteburn.sh
#!/bin/bash
while true;
    do openssl speed;
done
EOF

# 32 parallel 100% CPU tasks should hit even the biggest EC2 instances
for i in 1..32
do
    nohup /bin/bash /tmp/infiniteburn.sh &
done
```

Figure 2.17. BurnCPU script

snippets respectively related to a CPU fault (*BurnCPU*), a disk fault (*BurnIO*) and a network fault (*NetworkCorruption*).

In general, handling a virtual machine failure is often simple, since the failed virtual machine can be replaced with a new *healthy* virtual machine instance. Instead, it is difficult to detecting virtual machines that are only partially healthy (e.g., a service could randomly return faulty outputs, or the service could respond with a high latency). The *Latency Monkey* provides injection facilities to include this type of problems. For example, it can allow to inject delays in the RESTful client-server communication layer to simulate service degradation, node downtime or even service downtime.

Results. During 2012, Chaos Monkey has been applied to terminate over 65,000 instances running in Netflix production and testing environments, detecting many failure scenarios. Faults are tolerated most of the time, but developers continue to find issues caused by Chaos Monkey, allowing to isolate and resolve them so they don't happen again.

```
#!/bin/bash
# Script for BurnIO Chaos Monkey

cat << EOF > /tmp/loopburnio.sh
#!/bin/bash
while true;
do
    dd if=/dev/urandom of=/burn bs=1M count=1024 iflag=fullblock
done
EOF

nohup /bin/bash /tmp/loopburnio.sh &
```

Figure 2.18. BurnIO script

```
#!/bin/bash
# Script for NetworkCorruption Chaos Monkey

# Corrupts 5% of packets
tc qdisc add dev eth0 root netem corrupt 5%
```

Figure 2.19. NetworkCorruption script

2.5 Fault Injection Testing of Hypervisors

An important enabler to cloud computing systems are virtualization technologies. Think about most important clouds like Amazon's EC2, that uses Xen hypervisor, or IBM cloud-based systems that uses KVM hypervisor. The primary goal of hypervisors is to abstract and distribute computing resources between multiple VMs, leading to better resource utilization and flexibility (dynamic workload migrations). So, in order to have a high dependable cloud systems, it is desirable to have a solution providing a high-reliability support for virtualization and an accurate testing process for clouds.

2.5.1 CloudVal

CloudVal [125] is a framework to test the reliability of hypervisor within a cloud infrastructure. The framework provides an injector (implemented using debugger-based techniques) that allows to inject different type of faults like transient (soft) faults, guest misbehavior, performance faults and maintenance faults. This work is a starting point to develop a benchmark for validate cloud virtualization infrastructures.

The CloudVal framework supports fault injection in the KVM and Xen, both on the guest and on the host domains, and on the core modules of the hypervisors (i.e., *qemu-kvm* and the KVM kernel module for KVM [86]; *qemu-dm* and *xenstored* for Xen [18]). The tests are performed to evaluate VMs guest/host isolation and correlated hypervisor behavior, and the level of maintainability. Finally, *Virt-manager* [131] (a libvirt-based management system) is used by CloudVal for monitoring and managing a system during fault injection experiments.

Faultload. In this section is described the fault model used by CloudVal.

1. **Soft fault**: transient faults occurring in memory, CPU registers or

in the CPU control unit;

2. **Guest system misbehavior**: misbehavior of guest system by corrupting the state of a process (i.e., flip one or more bits related to stack, text area etc.) or raising CPU exception (e.g., machine check, divide by zero etc.);

3. **Performance fault**: process/thread delay, due to blocking operations, CPU overhead or interrupt events; these fautls can expose timing problems, such as race conditions;

4. **Maintenance fault**: situation where a specific hardware (e.g., CPU or memory bank) have to be turned off because of replacement or power management.

Architecture. As shown in Figure 2.20 CloudVal framework consists of three main components:

1. **Injector**: this component is implement as a loadable kernel module that resides on target machine. It is implemented using debugger-based techniques: (1) the injector sets a breakpoint in the code (trigger location) of the target component; (2) when the trigger location is executed, a breakpoint handler (that implements fault injection) is invoked; (3) a fault is injected, and the target components continues its execution;

2. **Control Host**: this component resides on a physical machine distinct from target machine. It provides all the facilities for testers, who specify all the parameters for a fault injection experiment (e.g., fault type and fault location) and generate scripts based on those, to perform the experiments. Furthermore, it collects logs and outputs from fault injection experiments;

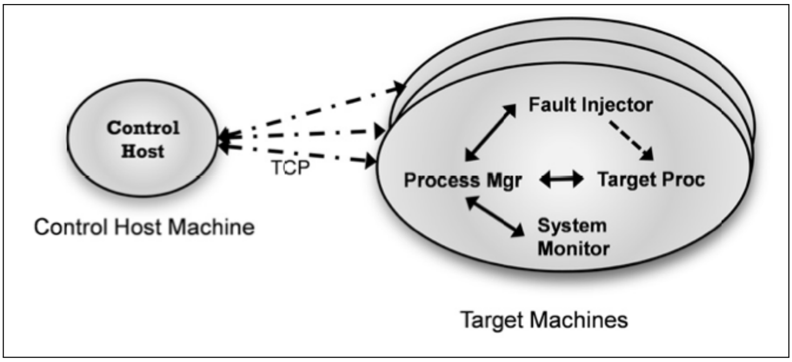


Figure 2.20. CloudVal architecture [125]

3. **Process Manager**: this component resides on the same machine (target machine) of the Injector component, and simply runs the commands received from the Control Host (e.g., trigger a fault) and sends back logs to the Control Host.

Results. Considering all fault type mentioned in subsection Fault Types, in Table 2.4 are reported the results. It is worth mentioning that CloudVal performs a pre-injection analysis to identify the fault triggers and fault locations. There are exploited two strategies: *stress-based*, that aims to inject faults into the most heavily used components of the target; *path-based*, that leverages on the sequence of instructions executed by the target program under a specific workload; periodically is read the *EIP* register[1], that contains the current instruction pointer, in order to get the locations on the execution path used as fault targets.

2.5.2 Xen failure mode analysis

Cerveira et al. [29] characterized Xen hypervisor failure modes by targeting the hybrid para-virtualization and hardware-assisted (PVH) mode.

[1] The *EIP* register always contains the address of the next instruction to be executed; this register can only be read through the stack.

Table 2.4. CloudVal results

Fault Type	**Hypervisor**	**Target**	# **Injected faults**	**Guest behavior**	**Hypervisor behavior**
Soft (in user space address)	KVM	Data Segment, stack segment and register of qemu-kvm process	500	Guest VM stopped when qemu-kvm crashed; furthermore, management system exhibits hang	120 qemu-kvm crashes. 26 become defunct (zombie) state. No kernel crash
	XEN	qemu-dm and xenstored	100	qemu-dm exhibits a crash, involving Guest VM to stop. Also xenstored exhibits a crash.	55 qemu-dm crashes. 12 defunct (zombie) state. No kernel crash. Due to those crashes, management system lose control of VMs
Soft (in kernel space address)	KVM	Data Segment, stack segment and register of KVM kernel module	1000	N/A	94 kernel crashes
	XEN	Crashing Dom0 by inserting a faulty kernel module	20	All Guest VMs stopped	20 kernel crashes
Guest system misbehavior	KVM	Code segment, data segment, stack segment and register of Guest OS kernel	14000	Guest kernel crashed	No fault propagation from guest to host system
	XEN	Code segment, data segment, stack segment and register of Guest OS kernel	30000+	2 cases in which the failure propagate to the host	N/A
Performance	KVM	Threads in qemu-kvm process	400	Guest system is not available during injected fault	16 kernel crashes, maybe due to activation of race conditions
	XEN	N/A	N/A	N/A	N/A
Maintenance (Turn OFF a CPU core)	KVM	Physical CPU core and CPU core in Guest VM	10	No effect	No effect
	XEN	CPU core in Dom0 and CPU core in Guest VM	10	No effect	No effect
Maintenance (Turn ON a CPU core)	KVM	Physical CPU core and CPU core in Guest VM	10	N/A when target is Physical CPU core and no effect when target is CPU core in Guest VM	10 kernel crashes only when KVM kernel module is loaded and when target is physical CPU core
	XEN	CPU core in Dom0 and CPU core in Guest VM	10	No effect when target is CPU core in Dom0; When target is CPU core in Guest VM, Guest kernel return error, but executes normally	Dom0 handles properly CPU on, rather than DomUs that does not turned back on CPU core (maybe due to incomplete implementation of the virtual CPU in Xen)

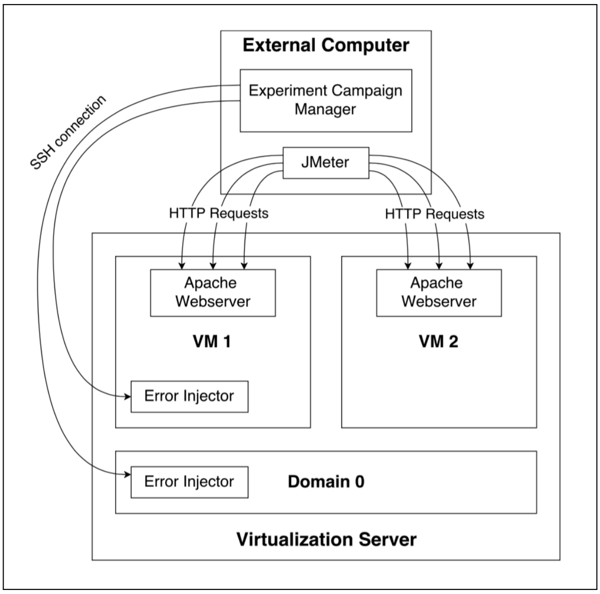


Figure 2.21. Diagram of the experimental setup. [29]

Exploiting the results obtained, the authors propose an approach based on fault injection testing to pinpoint how it is possible to evaluate the susceptibility of virtualization servers against soft errors, by comparing the PVH mode with the para-virtualization (PV) mode and the hardware-assisted virtualization mode (HV).

Faultload. The fault model used by the authors are bit-flips in the CPU registers. Such fault model is commonly accepted to be representative of soft errors that affect the CPU.

Architecture. As shown in Figure 2.21, the testbed consists of two VMs which run the Apache webserver; one of them (VM1) is the fault injection target, and the other VM (VM2) is running without any fault injection. These VMs are deployed on the Xen hypervisor, in which also the Dom0 component is subject to fault injection.

Results. The authors performed fault injection campaigns by targeting

(i) the applications running within the guest VM, (ii) the kernel processes (randomly chosen) within the guest VM, and (iii) the Xen- related processes of the Domain-0. Furthermore, the authors classified the results of the experiments according to the following:

- *Incorrect content*: The webserver produces syntactically correct HTML but the associated hash value is wrong;
- *Corrupted output*: The webserver provides an output that the client is unable to parse;
- *Connection reset*: The TCP connection between the client and the server is reset;
- *Client-side timeout.* The client triggers timeout because does not receive responses;
- *Hang*: The server does not respond to any request, and the client eventually triggers timeout;
- *No effect*: The injected fault does not have any impact on the system.

In particular the results show that:

- *Fault injection in application processes*: The faulty VM showed different failure modes. In the 84.6% of the faults did not have any effect in the output (the target register it was never used by the application or the application mask the fault). Furthermore, in the 12.9% of the faults, the client does not receive responses and trigger timeout. Regarding the isolation, both the fault free VM and the hypervisor were not affected by the injection;
- *Fault injection in guest VM kernel processes*: In this case, there are only little difference with injection in application processes, with a

higher percentage (98.6%) of faults with no effects. The authors explained such behavior by the fact that kernel processes are less likely to manifest the presence of soft errors. The isolation during that fault injection campaign was always guaranteed;

- *Fault injection in Xen processes in Dom0*: The results show that only *Hang* (of both VMs or of the hypervisor) and *No effect*failure modes are observed, respectively in the 60.2% and 39.8% of faults.

This page intentionally left blank.

Chapter 3

The Proposed Methodology

In this chapter we present the dependability benchmarking methodology for NFV. The methodology consists of three parts Figure 3.1. The first part consists in the definition of the *benchmark elements*, the *benchmark measures*, the *faultload* (i.e., a set of faults to inject in the NFVI) and the *workload* (i.e., inputs to submit to the NFVI) that will support the experimental evaluation of an NFVI. Based on these elements, the second part of the methodology consists in the execution of a sequence of fault injection experiments through the developed fault injection suite (see Chap. 4). In each fault injection experiment, the NFVI under evaluation is first configured, by deploying a set of VNFs to exercise the NFVI; then, the workload is submitted to the VNFs running on the NFVI and, during their execution, faults are injected; finally, the last step consists in collecting performance and failure data from the target NFVI at the end of the execution. At this point, the experimental testbed is cleaned-up (e.g., by un-deploying VNFs) before starting the next experiment. This process is repeated several times, by injecting a different fault at each fault injection experiment (while using the same workload and collecting the same performance and failure metrics). The execution of fault injection experiments can be sup-

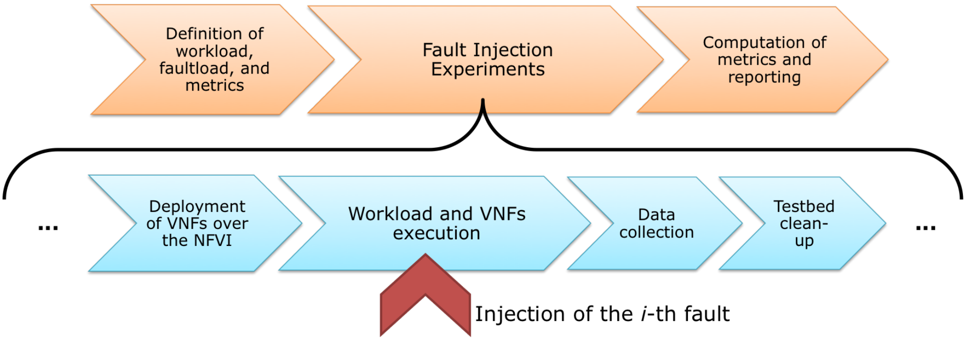

Definition of workload, faultload, and metrics
Fault Injection Experiments
Computation of metrics and reporting
...
Deployment of VNFs over the NFVI
Workload and VNFs execution
Data collection
Testbed clean-up
...
Injection of the *i*-th fault

Figure 3.1. Overview of dependability benchmarking methodology.

ported by automated tools for configuring virtualization infrastructures, for generating network workloads, and for injecting faults. Finally, performance and failure data from all experiments are processed to compute measures, and to support the identification of performance/dependability bottlenecks in the target NFVI.

In the following, we describe in details the elements that constitutes the methodology in NFV, according to its stakeholders and use cases (§ 3.1). Then, we address the problem of defining appropriate benchmark measures (§ 3.2), faultload (§ 3.3) and workload (§ 3.4) for NFV.

3.1 Benchmark Elements

In the ETSI NFV framework [111], the architecture of a virtualized network service can be decomposed in three layers, namely the *service*, *virtualization*, and *physical* layers (Figure 3.2). On the service layer, each VNF provides a traffic processing capability, and several VNFs are chained to provide added-value network services. Each VNF is implemented in software and runs on an NFVI, which includes virtual machines and networks deployed on physical resources. For example, Figure 3.2 shows a case in

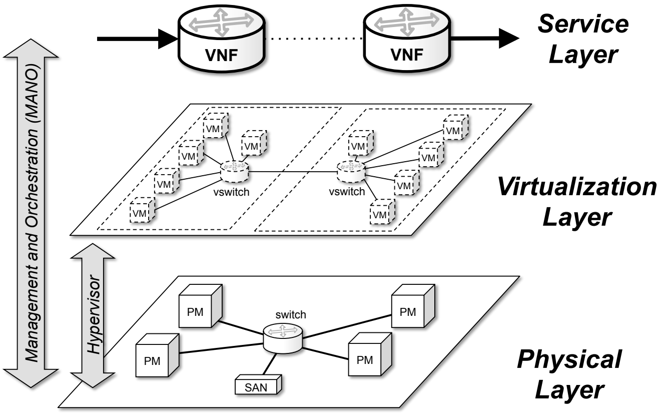


Figure 3.2. Architecture of Network Function Virtualization

which each VNF is implemented by a pool of VM service replica, by an additional VM to perform load balancing, and by a virtual network segment that connects the VMs. These VMs are scaled out and dynamically mapped to physical machines (PMs) in order to achieve high resource efficiency and performance. These operations are overseen by management and orchestration software (MANO) that installs and controls VNFs, and deploys them by using virtualization technology, such as hypervisors.

To identify the benchmark elements (SUB, BT, FIT), we need to consider the potential use cases of the dependability benchmark, which involve *telecom service providers*, *NFV software vendors*, *NFV infrastructure providers*, and *telecom service customers*, as users and performers of the benchmark:

1. A telecom service provider designs a network service, by composing a VNF service chain using VNF software, that can be developed in-house and provided by third-party NFV software vendors. The telecom provider performs the benchmark to get confidence that the network service is able to achieve SLOs even in worst-case (faulty) conditions. End-users and other telecom providers, which consume

network services on a pay-per-use basis (*VNFaaS*), represent the benchmark users: they can demand from the telecom provider empirical evidence of high dependability and performance, to make informed decisions based on both cost and dependability concerns. This evidence can be produced by the telecom provider in cooperation with an independent certification authority, in a similar way to certification schemes for cloud services [145, 57, 32].

2. An NFV infrastructure provider setups an environment to host VNFs on a pay-per-use basis (*NFVIaaS*) from telecom operators, which are the benchmark users. The NFVI is built by acquiring off-the-shelf, industry-standard hardware and virtualization technologies, and by operating them through MANO software provided by NFV software vendors. The NFVI provider performs the dependability benchmark to get confidence on the dependability of its configuration and management policies, with respect to faults of hardware and virtualization components. The NFVI provider revises the configuration and the MANO according to the feedback of the dependability benchmark.

In both use cases, the SUB must include the service layer, the infrastructure level (both virtual and physical), and the MANO. The BT is represented, respectively, by the VNF service chain, and by the MANO. In the former use case, the benchmark provides feedback on the robustness of VNF software, on the composition of VNFs into a chain, and on the configuration of the NFVI. In the latter, the benchmark provides feedback on fault management mechanisms and policies in the MANO. In both cases, the FIT is represented by the NFVI (both physical and virtualization layers). The NFVI is built from off-the-shelf hardware and virtualization components that are relatively less reliable than traditional telecom equipment, and thus represent a risk for the reliability of the NFV network as

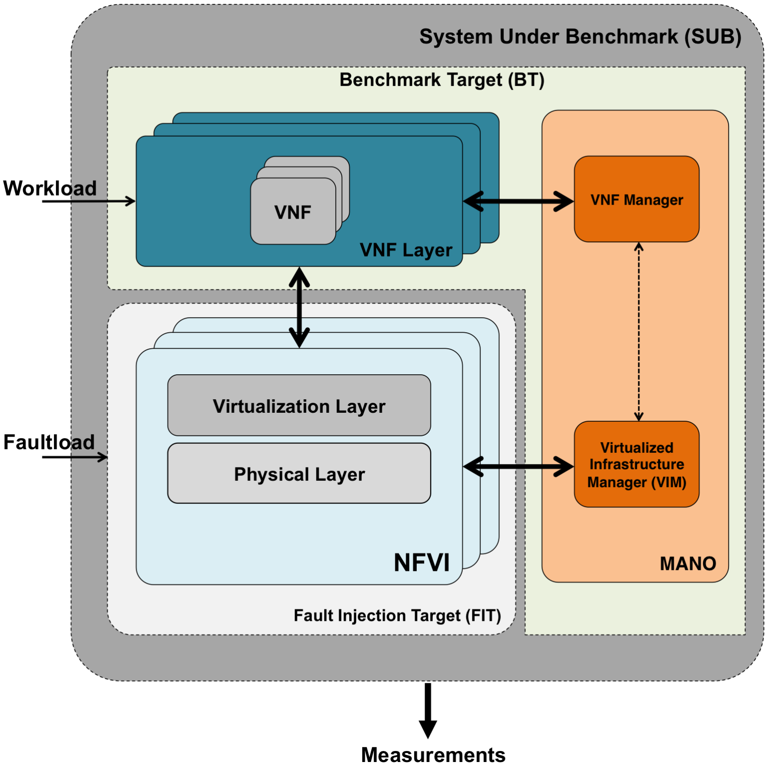


Figure 3.3. The benchmark target in NFV.

a whole. Thus, as we will discuss later, the benchmarking process injects faults in the NFVI, in order to assess the VNF service QoS in spite of faults in the NFVI, and to evaluate and tune MANO software with respect to these faults. Figure 3.3 shows the choice of the elements constituting the benchmark target.

3.2 Benchmark Measures

The measures are a core part of a dependability benchmark, since they represent the main feedback provided to the benchmark user. According

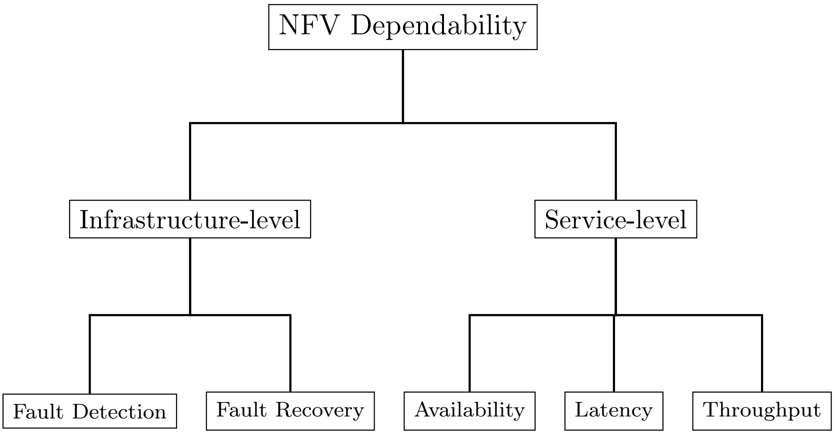


Figure 3.4. NFV dependability benchmark measures

to the two use cases of the previous section, we identify two groups of benchmark measures (Figure 3.4):

- *Service-level* (VNF) measures, which characterize the quality of service as it is perceived by VNF users, including measures of performance and availability of the VNF services;
- *Infrastructure-level* (NFVI) measures, which characterize the NFVI in terms of its ability to detect and to handle faulty conditions caused by hardware and software components inside the infrastructure.

3.2.1 Service-level measures

We define service-level measures according to the following driving criterion: We want to connect the results of the dependability benchmark to SLA (Service Level Agreement) requirements of the VNF services under benchmark. Typically, SLA requirements impose constraints on service performance in terms of latency and throughput, and on service availability in terms of outage duration.

The service-level measures of the benchmark include the *VNF latency* and the *VNF throughput*. It is important to note that, while latency and throughput are widely adopted for performance characterization, we specifically evaluate latency and throughput *in the presence of faults* in the underlying NFVI. We introduce latency and throughput measures to quantify the impact of faults on performance, and evaluate whether the impact is small enough to be acceptable. In fact, it can be expected that performance will degrade in the presence of faults, leading the system to exhibit higher latency and/or lower throughput, since less resources will be available (due to the failure of components in the NFVI) and since the fault management process takes time (at least few seconds in the case of automated recovery, and up to several minutes in the case of manual recovery).

▷ ***VNF Latency***. In general terms, network latency is the delay that a message "takes to travel from one end of a network to another" [124]. A similar notion can also be applied to network traffic processed by a VNF, or, more generally, by a network of interconnected VNFs (i.e., the *VNF graph* [113]), as in Figure 3.5. Latency can be evaluated by measuring the time between the arrival of a unit of traffic (such as a packet or a service request) at the boundary of the VNF graph, and the time at which the processing of that unit of traffic is completed (e.g., a packet is routed to a destination after inspection, and leaves the VNFs; or, a response is provided to the source of a request).

The *VNF latency* is represented by percentiles of the empirical cumulative distribution function (CDF) of traffic processing times. The CDF of service latency is given by:

$$L_e(x) = P(l_{i,e} < x)$$

where $l_{i,e}$ is the latency for the i-th traffic unit in the experiment e, which is defined as $l_{i,e} = t_{i,e}^{\texttt{res}} - t_{i,e}^{\texttt{req}}$, where $t_{i,e}^{\texttt{req}}$ and $t_{i,e}^{\texttt{res}}$ refer to the time of a request and of its response, respectively. In particular, SLAs typically

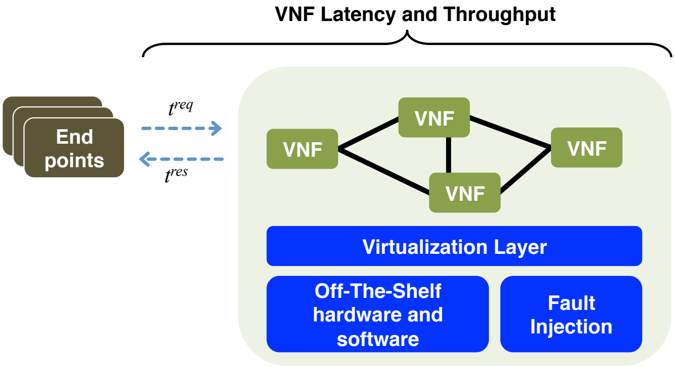


Figure 3.5. VNF Latency and Throughput.

consider the 50th and the 90th percentiles of the CDF (i.e., $L_e(50)$ and $L_e(90)$), to characterize the average and the worst-case performance of telecommunication systems [20]. The metric is computed from the time of fault injection is ended to the end of experiment.

▷ ***VNF Throughput***. In general, throughput is the rate of successful operations completed in an observation interval. The *VNF throughput* is represented by the rate of processed traffic (e.g., packets or requests per second) by VNF services in presence of faults (Figure 3.5). The VNF throughput over a period of time in an experiment e is given by:

$$T_e = \frac{N}{t_e^{\texttt{end}} - t_e^{\texttt{begin}}}$$

where $t_e^{\texttt{begin}}$ and $t_{i,e}^{\texttt{end}}$ refer to the start and the end time of the observation period, respectively, and N is the total number of traffic units (i.e., all traffic processed during an experiment e). The metric is computed from the time of fault injection is ended to the end of experiment.

The example in Figure 3.6 shows three potential cases of latency distributions: (i) latency in fault-free conditions, (ii) latency in faulty condi-

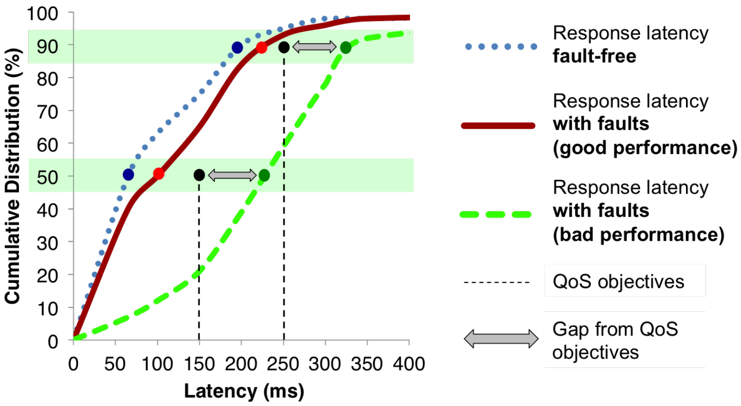


Figure 3.6. Examples of VNF latency distributions

tions, in which the network is still providing good performance, and (iii) latency in faulty conditions, in which performance is severely degraded. The 50th and the 90th percentiles are compared to *reference values* for these percentiles, which specify the maximum allowed value of the percentile for an acceptable quality of service (for instance, reference values imposed by SLAs). In the example of Figure 3.6, the maximum allowed values are 150ms for the 50th percentile, and 250ms for the 90th percentile (e.g., low-latency, interactive services, such as VoIP [5]). Both values are exceeded in the faulty scenario with performance degradation; in this case, the fault affects the performance perceived by VNF users.

The dependability benchmark aggregates VNF latency percentiles and throughput values from fault injection experiments, by computing the *number of experiments where latency percentiles and throughput exceeded reference values*, which points out the scenarios in which VNF performance is vulnerable to faults. Moreover, the *maximum value among latency percentiles* and the *minimum value among throughput values* provide a brief indication of how severe performance degradation can experience under faults. These aggregated values can be computed over the entire set of fault injection experiments to get an overall evaluation of NFV systems and compare them. Another approach is to divide the set of experiments into subsets, with respect to the injected faults, or with respect to the component targeted by fault injection, and then to compute aggregate values for each subset. Among these experiments, benchmark users are interested to know in which ones the latency percentiles and throughput *exceeded their reference values*, which point out the specific faults or parts that expose VNFs to performance issues.

Finally, we introduce the *VNF unavailability* to evaluate the ability of VNFs to avoid, or to quickly recover from service failures. Differing from latency and throughput, which assess whether faults cause performance degradation, this measure evaluates whether faults escalate into user-perceived outages. SLAs require that service requests must succeed with a high probability, which is typically expressed in *nines*. It is not unusual that telecom services must achieve an availability not lower than 99.999%–this implies that the monthly "unavailability budget" of VNF services amounts to few tens of seconds [20]. For example, the ETSI "*NFV Resiliency Requirements*" [117] reports that voice call users and real-time interactive services do not tolerate more than 6 seconds of service interruption, while the TL-9000 forum [130] has specified a service interruption limit of 15 seconds for all traditional telecom system services. Service disruptions of longer than a few seconds are likely to be considered service outages (as they impact not only isolated user service requests, but also the user retries of those failed requests), and thus accrue downtime and impact service availability metrics. Therefore, we introduce benchmark measures to evaluate the duration of VNF service outages.

▷ ***VNF Unavailability***. A VNF service is considered unavailable if either a unit of traffic is not processed within a maximum time limit (i.e., it is timed-out), or an error is signaled to the user as a response. The *VNF Unavailability* is defined as the amount of time during which VNF users experience this behavior (see Figure 3.7).

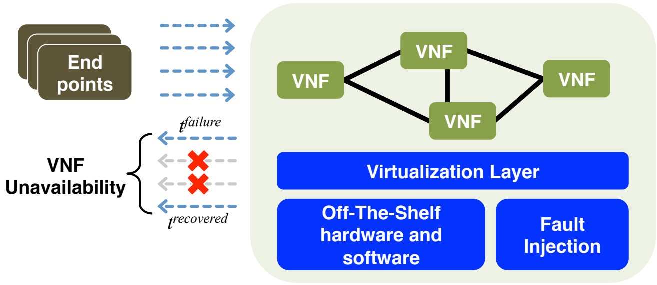


Figure 3.7. VNF Unavailability

During an experiment, after the injection of a fault, the VNF service may become unavailable (i.e., the rate of service errors exceeds a reference limit), and return available when the service is recovered. Moreover, a service may oscillate from availabile to unavailable, and viceversa, for several times during an experiment (e.g., there are residual effects of the fault that cause sporadic errors). Thus, the VNF Unavailability is given by the sum of the "unavailability periods" (denoted with i) occurred during an experiment:

$$U = \sum_i t_i^{\texttt{avail}} - t_i^{\texttt{unavail}}$$

where:

$$t^{\texttt{injection}} < t_i^{\texttt{unavail}} < t_i^{\texttt{avail}} < t^{\texttt{end}},\ \forall i$$

$$\text{error-rate}(t) > \text{error-rate}^{\texttt{reference}}, \forall t \in [t_i^{\texttt{unavail}}, t_i^{\texttt{avail}}], \forall i.$$

Figure 3.8 shows an example of the unavailability definition. If the VNF service does not experience any user-perceived failure, the VNF Unavailability is assumed to be $U = 0$. If the VNF service is unable to recover within the duration of the experiment (say, ten minutes), we conclude that the VNF cannot automatically recover from the fault, and that it needs to be manually repaired by a human operator. In this case, recovery takes orders of magnitude more time than required by SLAs: we mark this case by assigning $U = \infty$ to VNF Unavailability.

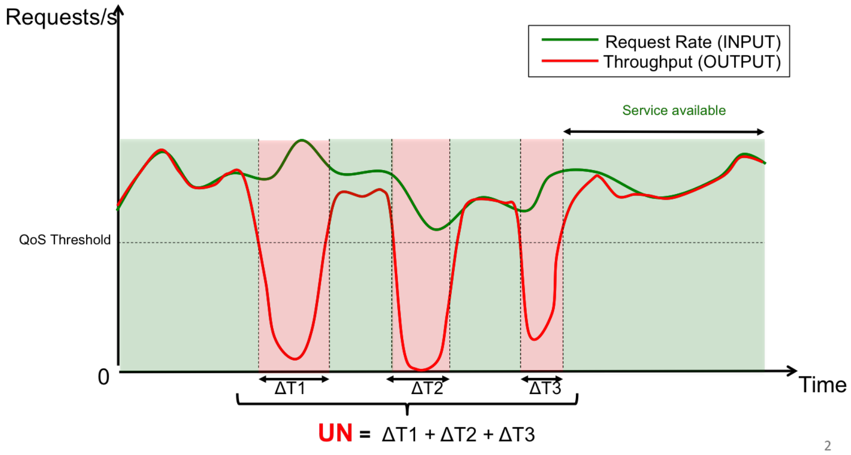


Figure 3.8. VNF Unavailability definition

The dependability benchmark aggregates VNF Unavailability values from fault injection experiments, by identifying the *experiments in which VNFs cannot be automatically recovered* (i.e., VNF Unavailability is a finite value), which points out the scenarios that need to be addressed by the NFV system designers. Moreover, the *maximum* and the *average* of *(finite) VNF Unavailability values* indicate how much the VNF is able to mask the faults. In a similar way to performance measures, the aggregated values of VNF Unavailability can be computed over subsets of fault injection experiments (e.g., divided by type or target of injected faults) for a more detailed analysis.

3.2.2 Infrastructure-level measures

Infrastructure-level measures are aimed at providing insights to NFVI providers on the fault-tolerance of their infrastructure. Several fault-tolerance solutions can be leveraged to this goal: the ETSI document on “NFVI Resiliency Requirements” [117] discusses the strategies that will likely be adopted in NFV, and our previous study [36] briefly summarizes them. These fault tolerance algorithms and mechanisms (FTAMs) are provided by the NFVI (e.g., at hypervisor or at middleware level), and VNF services should be designed and configured to take advantage of them. FTAMs be broadly grouped in two areas:

- **Fault Detection**: FTAMs that notice a faulty component (such as a VM or a physical device) as soon as a fault occurs, in order to timely start the fault management process. Examples of fault detection mechanisms are: heartbeats and watchdogs, which periodically poll the responsiveness of a service or of a component; performance and resource consumption monitors; internal checks performed internally in each component (such as the hypervisor) to report on anomalous states, failures of I/O operations and resource allocations; data integrity checks;

- **Fault Recovery**: FTAMs that perform a recovery action to mitigate a faulty component. Examples of recovery actions for NFVIs include the (de-)activation of VM replica and their migration to different hosts, by leveraging virtualization technologies. Moreover, VMs and physical hosts can be reconfigured, for instance, by updating a virtual network configuration, by deactivating a faulty network interface card, or by retrying a failed operation. The recovery action can succeed or not, depending on the ability of the VNF and of the hypervisor to maintain a consistent state after the recovery action (i.e., the VNF is able to work correctly after recovery). A fault is successfully recovered if the time-to-recovery is below a maximum allowed time, which depends on the type and criticality of a VNF, ranging from few seconds (e.g., 5 seconds) in the most critical scenarios, to tenths of seconds (e.g., 25 seconds) in the less critical scenarios. Moreover, it is required that VNF performance after recovery should be comparable to the performance of VNFs before the occurrence of a fault.

Since implementing and combining all these solutions in the fault management process is a complex task, we introduce benchmark measures to quantitatively evaluate their effectiveness.

▷ ***NFVI Fault Detection Coverage and Latency***. We define the *Fault Detection Coverage* (FDC) as the *percentage of fault injection tests in which the NFVI issues a fault notification*, either on individual nodes of the infrastructure (e.g., hypervisor logs), or on MANO software. If a fault is not detected, then a recovery action cannot be triggered. The FDC is computed by counting both the number of tests in which the injected fault is reported by the NFVI ($\#F_{\texttt{fault\_detected}}$), and tests in which the injected fault is not reported but causes service failures or performance degradations ($\#F_{\texttt{fault\_undetected}}$):

$$FDC = \frac{\#F_{\texttt{fault\_detected}}}{\#F_{\texttt{fault\_undetected}} + \#F_{\texttt{fault\_detected}}} \quad .$$

The *Fault Detection Latency* (FDL) is the time between the injection of a fault ($t_e^{injection}$), and the occurrence of the first fault notification ($t_e^{detection}$). The Fault Detection Latency is computed for the subset of experiments e in which a fault is actually detected. The FDL is given by:

$$FDL = t_e^{detection} - t_e^{injection}$$

▷ ***NFVI Fault Recovery Coverage and Latency***. The *Fault Recovery Coverage* (FRC) is the percentage of tests in which a recovery action (triggered by fault detection) is successfully completed. For example, in the case of a VM restart, the recovery is considered successful if a new VM is allocated, and VM software is correctly booted and activated. In the case of switching to redundant hardware resources, recovery is considered successful if enough redundant resources are available and they actually replace failed ones. The FRC is represented by the ratio between the number of tests in which faults were detected ($\#F_{\texttt{fault\_detected}}$), and the number of tests in which the recovery action is *successfully completed* ($\#F_{\texttt{fault\_recovery}}$):

$$FRC = \frac{\#F_{\texttt{fault\_recovery}}}{\#F_{\texttt{fault\_detected}}} \quad .$$

For those experiments in which the NFVI is able to perform a recovery action, we define the *Fault Recovery Latency* (FRL) as:

$$FRL = t_e^{recovery} - t_e^{detected}$$

where $t_e^{recovered}$ refers to the time when the NFVI concludes a recovery action.

3.3 Fault Model

A fault model defines a fault load, i.e., a list, and their relationships with system's components, of faults and exceptional conditions that are injected to emulate the real threats the system would experience during operation [44]. According to the DBench initiative [44], the definition of a "good" faultload is a tricky task, since the following properties need to be satisfied. The faultload needs to be *complete* and *representative*. This means that the defined faults should i) be representative of real-world operational faulty conditions, and ii) cover all the system's components. The faultload has to be*acceptable* by the benchmark stakeholders; it should be *portable* across different systems under benchmark, and should be *implementable* with reasonable efforts; it should be *adaptable*, that is, the benchmark users and performers should be able to selectively use only some parts of the faultload (e.g., to focus the benchmark on the fault-prone parts of the system, according to their knowledge of the target system).

We have defined the faultload to address the above properties by construction. To be portable across NFV implementations, and to cover the complete architecture of an NFVI [114], we define a fault model for the four domains of the NFV framework: *network*, *storage*, *CPU*, and *memory*. These elements are present both as *virtual* resource abstractions, and as *physical* resources of the NFVI (Figure 3.2). The fault model is aimed to the benchmark performer, which builds the faultload for the target NFVI, by applying the fault model to each resource in the NFVI (virtual and physical machines, virtual and physical disks and switches). For each physical and virtual machine in the NFVI, the faultload includes *physical and virtual CPU, memory, disk and network faults* for the physical/virtual CPUs, memory, disk interface and network interface of that machine. Moreover, for each virtual and physical storage and network element in the NFVI (virtual/physical disks and switches), the faultload is extended with *common-mode network and storage faults*, i.e., faults that affect, at the same time, all PMs or VMs connected to the physical or virtual disk or switch (Table 3.1).

In theory, once we identify the root causes of potential hardware or software problems, the process of injecting the fault (e.g., for hardware faults, for example, by adopting the bit-flip fault model, and for software faults, for example, by injecting code changes based on field data statistics about the frequency of fault types) it is on the one hand always applicable (highly portability), but on the other hand is extremely inefficient due to both the infinite potential ways for injecting such kind of faults, because we are injecting too close to the root cause. Moreover, we need to find faults that actually impact on the system, and such process has a non-negligible amount of time generally. These problems may lead the proposed fault model (and the methodology) to be not useful.

Conforming to the Laprie's et al. definitions [15] of *fault-error-failure* chain, in the scientific literature, as well as in the industrial practice, the injection of faults has been addressed using several methods. According to [110], in most cases fault injection is actually implemented through an indirect approach, by emulating the *effects* of hardware/software/configuration faults (i.e., the *errors* in the software state, provoked by the fault), instead of injecting the *actual* faults: such approach is named *error injection*. Error injection can significantly improve the feasibility and efficiency of fault injection, since: (i) the injection of the effects is less intrusive and technically easier to implement, as mimicking the actual faults (such as the physical electromagnetic interferences) can be cumbersome, difficult to automate, and even destructive for the system under test; (ii) the injection of the effects avoids to wait the time required for fault activation and error propagation, which may be quite significant (e.g., "long latency" faults that stay latent for most of the experiment duration) and may even not be activated/propagated at all during the (necessarily limited) period of the experiment (since the activation/propagation is not directly controlled by the tester). For these reasons, error injection has gained popularity in the last decades. Moreover, error injection is today accepted as an accurate-enough approximation of hardware faults, such as bit-flips for CPU faults [125] and data corruption/delays for network and disk I/O [77], and more recently it has been adopted to also emulate the effects of software faults [87].

In the context of NFV, the definition of a comprehensive fault model could be very challenging. Furthermore, applying only one fault model (e.g., bit-flip model) for each component within the NFV framework can be not meaningful due to the different functionalities they provide towards the system. As mentioned before, the definition of a fault model by considering actual faults has the advantage of portability, but the problem of efficiency, i.e., the injected faults have to always trigger an error/failure. Instead, considering errors/failures in the faultload, can speed up and make more easy the definition of the fault model from a practical point of view; however, the fault model obtained does not give any guarantee for the portability because failures of components heavily depend from the internals of the target component itself.

To overcome these issues, according to the elements of the benchmark defined in the Sec. 3.1, we consider the "boundaries" that separate the NFVI (i.e., the fault injection target) from the benchmark targets: the boundary is the specific *interface* exposed by the NFVI components. It is important to note that an interface is not necessarily a physical one, but may be an abstraction that identifies a point of interaction between the target component and the rest of system. The fault model is obtained by identifying, for each target component, its interfaces and the possible effects of faults on these interfaces (errors, from the perspective of the system under benchmark). We consider three general types of errors (as discussed later in this section): unavailability, corruption, and delay of the component. To refine these errors in a more specific way for each component, we look for potential faults (either from software, from hardware, and from human operators) that may happen inside the component, and that may cause errors of these three general types. To identify the faults, we studied the literature and the experience reported by practitioners about failures in the field.

CPU and Memory Fault Model. CPU and memory hardware devices are affected by the well-known *soft errors*. In fact, during system operation, radiation particles (e.g., alpha particles and atmospheric neutrons), and power spikes lead to a signal or datum wrong in semiconductor devices. The classical fault model to assess CPUs and memories is by injecting *bit-flip* in main memory areas, and/or registers. In literature, different tools leverage the bit-flip fault model to emulate hardware faults for CPU and memory [19, 81, 84, 83, 150, 69, 28, 132, 41].

In the case of the CPU management subsystem (either virtual or physical), the *interface* towards the BT is represented by two basic CPU functionalities: (i) create and schedule execution flows (e.g., virtual machines, processes), and perform the operations of these execution flows (e.g., fetch and execute instructions, update the CPU registers, etc.). This interface allows user-space applications and OSs (e.g., guest OSs, hypervisors) to run. Thus, errors introduced at this level hinder the execution of these components on virtual and physical CPUs.

A similar reasoning can be done for *memory* subsystem. The system allocates memory areas for reading and writing data by using the interface the memory subsystem provide. Errors in that interface could threaten the functionalities of the memory subsystem, and may expose the memory to a high rate of swapping, overloads, corruptions, and other deviations from normal service. In these cases, the memory subsystem does not allow new allocations, or even lead the system to crash.

In the proposed fault model, we consider the interfaces that exports both the CPU and memory, in order to apply the desired effect for the target component. For example, we emulate hardware faults by using the bit-flip fault model (e.g., corruptions of CPU registers and memory locations), and configuration faults by inject resource exhaustion failures (e.g., process hog that runs on the CPU, process that induces high memory swap rate).

I/O Fault Model. In literature, there are different studies that evaluate the storage subsystem reliability, and the related fault model. *Talagala et al.* [146] analyze which are faults that occur more frequently in storage systems. Over an 18-month period, she found that the majority of failures affecting disks are media errors, write failures, hardware errors (e.g., device diagnostic failures), SCSI timeouts, and SCSI bus errors. Based on these findings, Brown et al. [23] evaluate the availability of software RAID systems by injecting a variety of faults, such as media errors on reads/writes, hardware errors on SCSI commands, power failures, disk hangs.

For the evaluation of network subsystem, in [142] the authors leverage the bit-flip fault model to inject faults into the NIC processor. The experiments lead to different errors, such as dropped or corrupted packets, NIC resets, and interface hangs. Another study [46] evaluates the robustness of TCP network protocol in spite of typical network faults, which include dropping, delaying, reordering, and duplicating network packets among others. Moreover, *Fummi et al.* [60] evaluate the dependability of networked embedded systems, by leveraging more or less the same fault model as the study mentioned previously.

In the proposed fault model, the *interface* between the FIT (i.e., the I/O stack, including the NIC for networking and the HBA for storage) and the BT is represented by the *device driver*, which is the first software component that handles the communication between the device and the rest of the system. Thus, by introducing anomalies at the interface level, we can emulate hardware problems (faults) originated at the hardware level (e.g., the NIC processor is flawed), but also software-related bug in the context of the kernel OS (e.g., device driver bugs, kernel bugs), as well as configuration faults more related to the performance provided by the physical controller.

At this point, for each domain (physical and virtual CPU, memory, disk, and network), we identify what to inject according to the following three general fault types:

- *Unavailability*, the resource becomes unresponsive and unusable;
- *Delay*, the resource is overcommitted and slowed-down;
- *Corruption*, the information stored or processed by the resource is invalid.

These fault types are broad classes that span over the possible failure modes of a component [126, 106, 19, 38, 73]. We specialize these general fault types for each resource, by analyzing how hardware, software, and/or operator faults can likely cause these three possible fault types [44]. In this analysis, we consider the scientific literature on fault injection and failure analysis in cloud computing infrastructures (see Sec. 2.2), well-known cloud computing incidents [11] , [62], [161], and knowledge on the prospective architecture and products for NFVI (e.g., VMware ESXi, Docker containers) [111, 134], to identify a representative, complete, and acceptable set of faults. Moreover, these faults are feasible to implement using established software fault injection techniques [110].

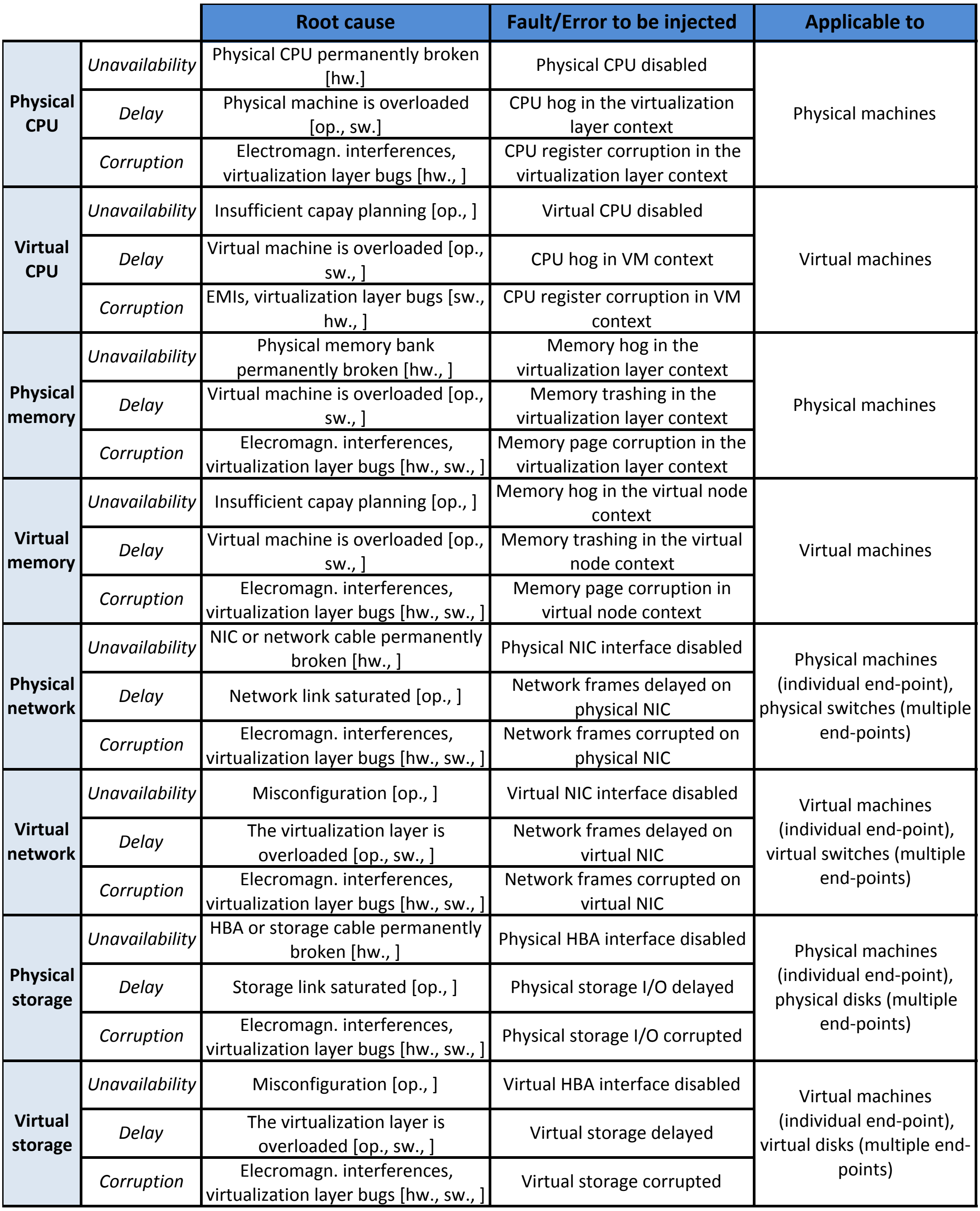
The fault mode has been summarized in Table 3.1, which shows, for each domain and for each fault type, the possible causes of faults, and the fault effects to be injected. Moreover, the table shows whether the fault model applies to physical and virtual machines, and to physical and virtual disk and network elements (which affect several physical/virtual machines at the same time). Furthermore, we adopt the most used mechanisms in literature to emulate faults in the CPU and memory subsystems [19, 81, 84, 83, 150, 69, 28, 132, 41], network subsystem [142, 46, 60], and storage subsystem [146, 23].

According to the criteria mentioned above, we identified the following fault models:

- **Physical CPUs and memory**: These physical resources can become abruptly broken due to wear-out and electrical issues. If these faults are detected by machine checks, they lead to CPU exceptions and to the de-activation of failed CPU and memory banks. Otherwise, these faults cause silent corruptions of random bytes in CPU registers and memory banks; even in the case of ECC memory, data can become corrupted before being stored in memory, when flowing through the CPU or the bus. Software faults in the virtualization layer (e.g., the VMM (Virtual Machine Monitor, such as a hypervisor)) may cause the corruption of whole memory pages, due to buggy memory management mechanisms (such as transparent page sharing and compression) or to generic memory management bugs at virtualization layer level (such as buffer overruns and race conditions). Moreover, physical CPUs and memory can be overloaded by an excessive number of VMs, or by buggy services that run in the virtualization layer; in turn, the contention on CPUs and memory may lead to scheduling and allocation delays of VMs;

- **Virtual CPUs and memory**: The VMs may lack virtual CPUs and virtual memory due to optimistic resource reservations to the VMs. In a similar way to physical CPUs and memory, electrical

Table 3.1. Overview of the fault model.



		Root cause	Fault/Error to be injected	Applicable to
Physical CPU	*Unavailability*	Physical CPU permanently broken [hw.]	Physical CPU disabled	Physical machines
	Delay	Physical machine is overloaded [op., sw.]	CPU hog in the virtualization layer context	
	Corruption	Electromagn. interferences, virtualization layer bugs [hw.,]	CPU register corruption in the virtualization layer context	
Virtual CPU	*Unavailability*	Insufficient capay planning [op.,]	Virtual CPU disabled	Virtual machines
	Delay	Virtual machine is overloaded [op., sw.,]	CPU hog in VM context	
	Corruption	EMIs, virtualization layer bugs [sw., hw.,]	CPU register corruption in VM context	
Physical memory	*Unavailability*	Physical memory bank permanently broken [hw.,]	Memory hog in the virtualization layer context	Physical machines
	Delay	Virtual machine is overloaded [op., sw.,]	Memory trashing in the virtualization layer context	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Memory page corruption in the virtualization layer context	
Virtual memory	*Unavailability*	Insufficient capay planning [op.,]	Memory hog in the virtual node context	Virtual machines
	Delay	Virtual machine is overloaded [op., sw.,]	Memory trashing in the virtual node context	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Memory page corruption in virtual node context	
Physical network	*Unavailability*	NIC or network cable permanently broken [hw.,]	Physical NIC interface disabled	Physical machines (individual end-point), physical switches (multiple end-points)
	Delay	Network link saturated [op.,]	Network frames delayed on physical NIC	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Network frames corrupted on physical NIC	
Virtual network	*Unavailability*	Misconfiguration [op.,]	Virtual NIC interface disabled	Virtual machines (individual end-point), virtual switches (multiple end-points)
	Delay	The virtualization layer is overloaded [op., sw.,]	Network frames delayed on virtual NIC	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Network frames corrupted on virtual NIC	
Physical storage	*Unavailability*	HBA or storage cable permanently broken [hw.,]	Physical HBA interface disabled	Physical machines (individual end-point), physical disks (multiple end-points)
	Delay	Storage link saturated [op.,]	Physical storage I/O delayed	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Physical storage I/O corrupted	
Virtual storage	*Unavailability*	Misconfiguration [op.,]	Virtual HBA interface disabled	Virtual machines (individual end-point), virtual disks (multiple end-points)
	Delay	The virtualization layer is overloaded [op., sw.,]	Virtual storage delayed	
	Corruption	Elecromagn. interferences, virtualization layer bugs [hw., sw.,]	Virtual storage corrupted	

issues and VMM bugs may lead to data corruptions in VM context. Finally, software and operator faults inside the VM may overload virtual CPUs and memory;

- **Physical storage and network**: Storage and network links (respectively, HBA and NIC interfaces, and connections between machines, and network switches and storage) may fail due to wear-out and electrical issues. Moreover, electrical issues and software bugs in device drivers may cause the corruption of block I/O and network frames. The storage and network bandwidth may get saturated due to excessive load and insufficient capacity planning, causing the overall delay of I/O traffic;

- **Virtual storage and network**: Storage and network interfaces of individual VMs, and virtual switch and storage connections, may become unavailable due to human faults in their configuration. In a similar way to physical storage and network, wear-out and electrical issues may affect the I/O traffic of specific VMs, and the I/O traffic may be delayed due to bottlenecks caused by the emulation of virtual switches and storage.

It is important to note that the proposed fault model can be adapted to the specific target NFVI. In particular, software faults in the virtualization layer (in particular, bugs in device drivers and in other extensions to the virtualization layer developed by untrusted third-parties) may be omitted depending on the integrity and maturity of this software. Overloads may be omitted in the case that resources are over-provisioned, or in the case that capacity planning has been carefully performed. Operator faults may be omitted in the case that configuration of the virtualization layer is fully automated, and configuration policies are carefully checked. The fault model allows removing specific faults according to their root causes;

another possible approach is to give different weights to faults when evaluating the experimental results of the benchmark [37].

3.4 Workload

The definition of the workload includes the identification of how to exercise the system during the faultload is exercised. In order to obtain reasonable and realistic results from dependability benchmark, these workloads should be representative of what usually the NVF will face once deployed in a real scenario. Typical workloads in the network domain are network traffic that follows a specific pattern, such as staircase load pattern or a flat load pattern with ramp up, and so on [118]. A realistic workload can be automatically generated by using existing load generators or benchmarking tool. Note that the selection of workloads also depends on the kind of network service that is hosted on the NFVI. For these reason, we refer the reader to existing network performance benchmarks and network load generators. Suitable examples of workloads for NFVIs are represented by performance benchmarks specifically designed for cloud computing systems [34, 22, 137] and by network load testing tools such as Netperf [70].

This page intentionally left blank.

Chapter 4

Fault Injection Tool Suite

4.1 Virtualization Technologies Background

Virtualization is the cornerstone technology of NFV. Indeed, it allows to abstract the specific details of physical resources (e.g., CPUs, network and storage devices, etc.), providing and sharing them for applications at higher levels. This perspective completely changes the way we see a physical machine (or a pool of physical machines), making it a simply soft component to use and manage at the push of a button. A main concept within virtualization is the *Virtual Machine* (VM), which provides an isolated execution context running on top of the underlying physical resources. The *Hypervisor* actually runs VMs and it is also responsible of coordinating multiple VMs in order to discipline the access to the underlying CPU, memory and I/O resources.

Currently, in the virtualization panorama there is a spectrum of approaches, but mainly *Hypervisor-based* and *Container-based* (or Operating system-level) virtualization are actually in use.

Hypervisor-based virtualization technologies provide an environment that allows for a guest operating system to run on top of a so-called Hypervisor. As mentioned above, hypervisor is a piece of software that runs multiple virtual machines, each of which run a guest operating system. There are two main types of Hypervisor-based virtualization:

- *Full virtualization*, in which the hypervisor completely isolates the guest OS and completely abstracts hardware resources to guest. The task of the hypervisor is to emulate privileged CPU instructions and I/O operations, and to multiplex resources between concurrent VMs. Examples of hypervisors with full virtualization are VMware ESXi [155], KVM [86], and Microsoft Hyper-V [102].

- *Paravirtualization* (partial virtualization), in which the guest OS is aware that it is running in a VM, that is, it is modified in order to communicate directly with the hypervisor (via a system call mechanism called *hypercalls*) to achieve better performance. An example of hypervisor with paravirtualization is Xen [17].

On the other side of spectrum, **Container-based** virtualization, also called *Operating System-level* virtualization, allows to runs multiple guest OSes without hardware virtualization. OSes have been already designed to provide resource isolation, but among processes. Thus, the idea is to use a traditional OS to run virtual appliances, by enhancing the (host) kernel OS to provide isolation between guest applications that run in so-called *containers*. A container is not a Virtual Machine in the traditional sense, but it is an environment that allows to runs a guest application with its own filesystem, memory, devices, processes, etc., leveraging kernel OS host process isolation (e.g., *namespace* in Linux [94]) and resource management capabilities (e.g., *cgroups* in Linux [98]). Examples of container-based virtualization technologies are LXC (LinuX Container [95]), Docker [49], OpenVZ [122].

In the following, we deeply analyze two alternative virtualization technologies: VMware ESXi and Docker, respectively an hypervisor-based and a container-based solution.

4.1.1 The VMware ESXi Architecture

To identify the failure modes of ESXi virtualization mechanisms, we first need to analyze their architecture. Some of the virtualization techniques developed by VMware have been described in research papers [160, 7, 26]; additional information on VMware ESXi has been obtained from white papers on VMware products, and from documentation for developers (such as the device driver DDK documentation).

VMware has developed two main virtualization architectures over time. The oldest architecture (Figure 4.1) is based on standard x86 hardware and full emulation of devices, while the newest architecture (Figure 4.2) takes advantage of virtualization extensions in modern CPUs (**hardware-assisted virtualization**) (i.e., Intel VTx [152], and AMD-V [6]), and of virtualization-aware guest OSes (**paravirtualization**).

The traditional architecture

In the original architecture of VMware products (Figure 4.1), x86 virtualization did not rely on hardware virtualization extensions, but was implemented using x86-to-x86 code translation (to emulate virtualization-sensitive CPU instructions in software) and traditional x86 segmentation/pagination mechanisms (to virtualize memory). Moreover, the original architecture was aimed at executing unmodified commodity guest OSes (i.e., not aware of virtualization), by fully emulating I/O devices.

A key virtualization component is represented by the VMM (**Virtual Machine Monitor**). For each VM executing on an ESXi host, there exists a VMM, whose responsibility is to virtualize CPU and memory. The VMM mediates between the VM and the hardware CPU, by intercepting **virtualization-sensitive** instructions (i.e., instructions that change the

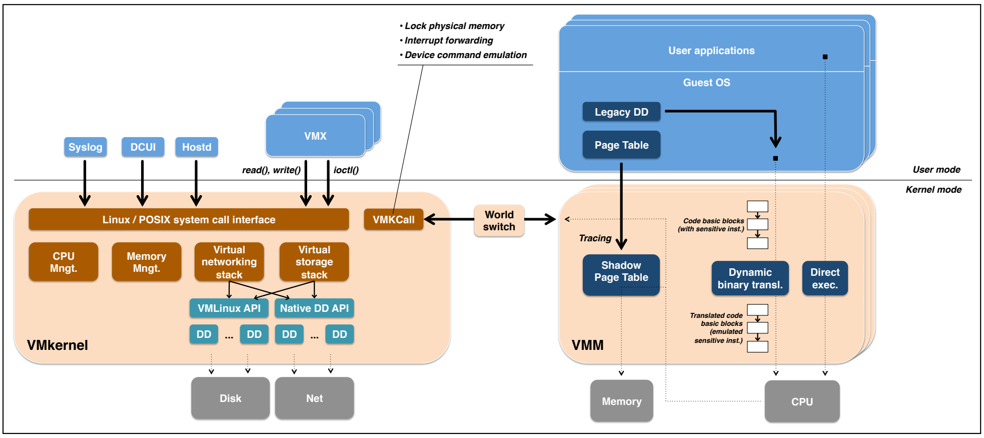


Figure 4.1. ESXi virtualization architecture using a traditional setup (no CPU virtualization extensions, no I/O paravirtualization).

configuration of physical resources). These instructions are emulated in software, thus providing virtualized resources to the VM, and allowing more than one VM to execute on the same hardware. Virtualization-sensitive instructions include instructions for setting privileged CPU flags (such as enabling/disabling interrupts), I/O instructions, segment/page table management, etc..

To perform these operations, the VMM takes full control of the CPU and of its registers. To do so, the VMM acts as a "peer OS" of the VMkernel. A VMM and the VMkernel alternate in the management of the CPU, by performing a **world switch** (i.e., a context switch involving every CPU register, including privileged ones). The VMkernel first instantiates a VMX process (running on the VMkernel side, and representing the VM); then, the VMX generates a VMM running in parallel with the VMkernel. The CPU is still scheduled by the VMkernel: when the VMX is scheduled, the VMX triggers a world switch and the execution of the VMM, using the **ioctl** system call. In general, the VMX interacts with the VMkernel and the VMM using **ioctl**, in order to initiate, configure, and schedule the VMM, and to exchange I/O device commands and data with a legacy device driver running on the guest OS.

In the original VMware architecture, the VMM emulates virtualization-sensitive instructions using **dynamic binary instrumentation**: guest OS code is translated block-by-block, by replacing sensitive instructions with their emulated counterparts. In particular, I/O instructions (issued by a legacy device driver in the guest OS) are handled by a **legacy I/O device emulator** running int the VMX process (which runs on the VMkernel in VMware ESXi, and on the host OS in VMware Workstation). In turn, the VMX issues regular system calls (e.g., *read*, *write*) on the VMkernel to execute I/O operations. VMware developed advanced optimizations such as code block caching, adaptive translation, partial evaluation, direct execution of user code, etc., to speed up code translation and to achieve performance comparable to non-virtualized execution.

The VMM uses a VMKCall interface to send requests for locking memory pages by the VMkernel (in order to allocate them for the virtual machines), to forward interrupts to the VMkernel (i.e., interrupts from physical I/O devices that occur during the execution of the VMM, but that must be managed by the VMkernel), and to send requests for emulating I/O commands of virtual I/O devices to the VMX (only in the case of the traditional VMware architecture, were legacy I/O devices are emulated by the VMX).

To virtualize memory, the VMM introduces a **shadow page table**. The shadow page table mirrors the page table of the guest OS, where virtual memory addresses are translated into physical memory addresses actually allocated to the VM. The shadow page table is the one actually used by the CPU to access memory. Every time that the guest OS changes its page table (i.e., writing "*guest-physical* page addresses"), the changes are intercepted by the VMM and propagated to the shadow page table (writing "*host-physical* page addresses"). To intercept such changes, the

VMM sets the guest OS page table as read-only, thus causing an exception at each change, which is handled and traced by the VMM.

The physical I/O resources, most notably storage and network, are managed by the VMkernel. In turn, the virtual networking and storage stacks of the VMkernel include several components. The most important components of the virtual storage stack are the *VMFS* filesystem (for managing VM datastores and *vmdk* virtual disks), the *Pluggable Storage Architecture* (for managing physical I/O paths in storage area networks), and the *storage device drivers* (for managing *iSCSI* and *FibreChannel* storage). The most important components of the virtual network stack are the *vSwitch* (for switching network frame to/from VMs and the physical network) and the *network device drivers* (for accessing to physical networks, such as *Gigabit Ethernet*). The VMkernel supports two APIs for developing and running device drivers: (i) the *VMLinux* API, for running legacy device drivers from the Linux OS; and (ii) the *Native* device driver API, used by device drivers specifically developed for the VMkernel.

Finally, a set of user-space processes (besides the VMXs) are executed on the ESXi. These processes are intended for remote management and monitoring purposes. Among them, the most relevant user-space processes are *hostd* (to provide a programmatic API interface to remote clients), *DCUI* (an user interface displayed on the ESXi host console), and *syslog* (for logging). These processes run on top of the VMkernel, and execute POSIX system calls to interface with the VMkernel, in a similar way to a Linux system.

The modern architecture

In the modern VMware architecture (Figure 4.2), the ESXi hypervisor leverages novel CPU virtualization extensions recently introduced by Intel and AMD, to further improve the performance of x86 virtualization.

These extensions avoid the need for dynamic binary translation. Using

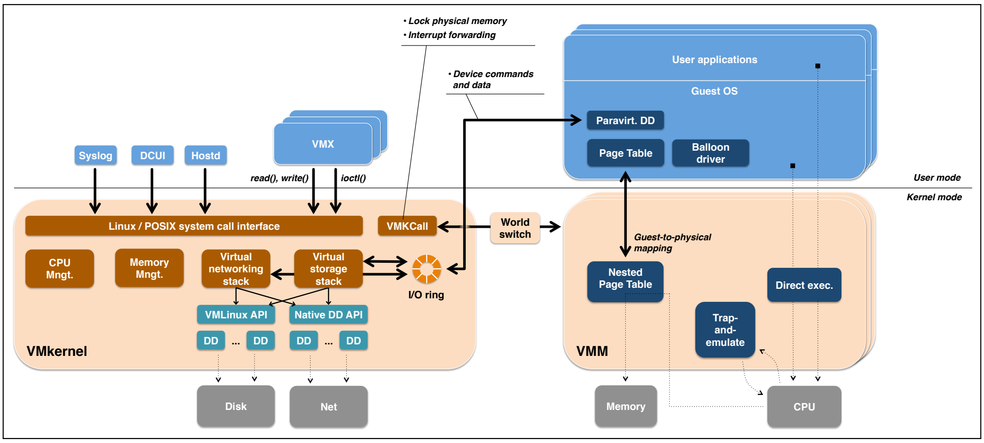


Figure 4.2. ESXi virtualization architecture, using a modern setup (CPU virtualization extensions, I/O paravirtualization).

these extensions, the VMM can instruct the CPU to **trap-and-emulate** sensitive instructions. When a sensitive instruction is executed, the CPU generates an exception, and the trapping instruction is emulated by the trap handler, which runs in VMM context.

Virtualization extensions also avoid the need for shadow tables, by introducing **nested page tables**. The nested page table is managed by VMM, and specifies guest-memory-to-host-memory mappings (i.e., where a page of virtual memory should be assigned to physical memory). The guest OS handles its page table to manage the memory of the VM. When the guest OS or applications access to VM memory, the CPU uses the nested page tables to automatically translate (in hardware) guest addresses into host addresses.

Furthermore, a **balloon driver** is loaded into the guest OS, in order to allow the VMkernel to reclaim memory from the virtual machine: when the ESXi host runs out of physical memory, the VMkernel can request to the balloon driver to pin virtual memory (by performing memory allocations on the guest OS), which becomes again available for use to the ESXi host. If even more memory is needed, the VMKernel performs **memory compression** and **hypervisor swapping**. The first mechanism consists in compressing pages whose compression ratio is greater than 50%, and de-compress those pages on demand. The area for storing compressed pages is not fixed, but changes dynamically depending on workload conditions. Hypervisor swapping allows the hypervisor to directly swap out guest-physical memory to a swap area on the disk.

As in the case of the traditional architecture, the physical I/O resources are managed by the virtual networking and storage stacks. To make I/O more efficient, VMware ESXi introduced the support to **I/O paravirtualization** (Figure 4.2). The guest OS in the VM runs a "paravirtualized" device driver (such as **pvscsi** and **vmxnet**), that is, a device driver **aware of I/O virtualization**. The paravirtualized device drivers interact with the VMkernel using a **shared I/O memory ring**. The VMkernel asynchronously consumes commands from the ring, and exchanges input and output data with the VM through the ring. The VMkernel accesses to the I/O rings through asynchronous kernel-level threads (**bottom-halves**), that are the intermediary between the network/storage stacks and the guest OSes. This approach reduces world switches and I/O latency, and avoids the need for the emulation of legacy I/O devices. This approach requires to install paravirtualized device drivers in the virtual machine, which are today easily obtained for the most important commodity OSes (such as Linux and Windows).

The modern architecture (CPU virtualization extensions and I/O paravirtualization, see Figure 4.2) is gaining widespread adoption, and will be the most used one in the near future. VMware ESXi is able to automatically select the best configuration (among the traditional architecture and the modern architecture) based on the availability of CPUs with virtualization extensions, and on the preferences of the system administrator.

High Availablity in VMware

The *VMware HA* technology is the main tool in VMware world to provide high availability for virtual machines and physical host. VMware HA adopts a mechanism based on heartbeats, in order to be able to continuously monitor all virtualized servers and to detect faults in the physical resources and failures of the VMs. VMware HA guarantee high availability for VMs through a primary host, which maintains the cluster state, and when needed it starts failover actions. VMware HA agents communicate with each other to detect failures and host network partition. The communication occurs via heartbeats mechanism. In the case of loss of heartbeats from the ESXi host, VMware HA automatically migrates and restarts the affected VMs on an available host in the resource pool [156]. Furthermore, in case of loss of heartbeats from VMs, VMware HA automatically restart them.

4.1.2 Container-Based Virtualization

Container-based virtualization, also called *Operating System-level* virtualization, allows to run multiple appliances without hardware virtualization. Resource virtualization in OSs is an old idea, based on the concept of process. The idea behind container-based virtualization is to enhance the abstractions of OS processes (so-called *containers*), by extending the (host) OS kernel. In container-based virtualization, a container has its own virtual CPU and virtual memory (like in traditional OS processes). In this kind of virtualization, it is also added abstractions for a virtual filesystem (i.e., the container perceives a filesystem structure that is different than the host's), virtual network (i.e., the container sees a different set of networking interfaces), IPC, PIDs, and users management. These virtual resources are distinct for each container in the system.

A container is not a VM in the traditional sense, since there is no emulation of physical devices (e.g., NICs, disk HBAs), CPU and memory. For this reason, this kind of virtualization is more *lightweight* than full- and para-virtualization. Container-based virtualization leverages the process abstractions (e.g., *namespace* in Linux [94]) and resource management capabilities (e.g., *cgroups* in Linux [98]) of the host kernel. The idea behind OS-level virtualization has been around since 80s, when was introduced the *chroot* system call as a first form of virtualization. Later, there was developed other OS-level virtualization technologies like Jail FreeBSD [80], and Solaris containers [129], but such approach has gained a big momentum during recent years. Currently, the most used container-based virtualization technologies are LXC (LinuX Container [95]), Docker [49], and OpenVZ [122]. Figure 4.3 depicts the container-based approach.

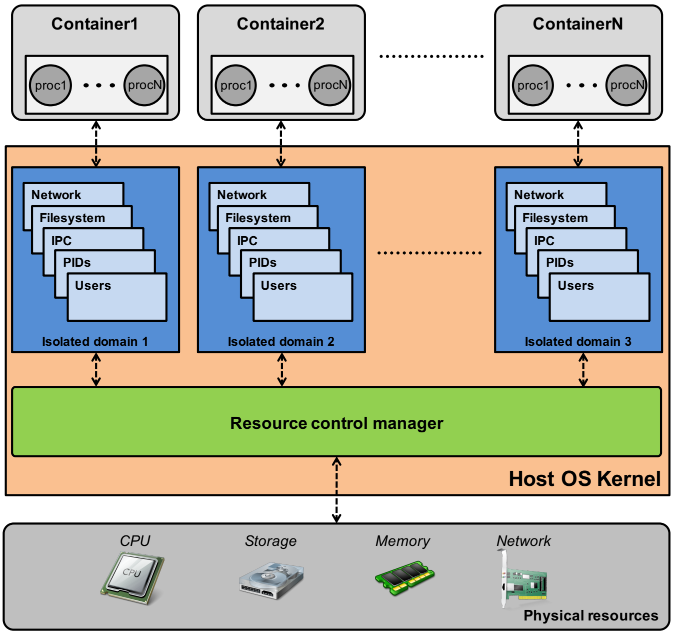


Figure 4.3. Container-based architecture

We are witnessing an increasing use of container-based virtualization in cloud infrastructures. Examples are Google [63, 65], Amazon [9], and Microsoft [101], which are already providing cloud services running on containers. Containers are expected to be fast, thus providing **high performance**, since there is no extra overhead due to emulation of devices. Moreover, containers make virtualization more **manageable**, since creating and moving containers is easier and faster.

In the following, is given a background on *namespaces* and *cgroups*, which are the main blocks of currently used container-based virtualization technologies, e.g., LXC, and, Docker.

Namespaces

Container-based virtualization is built upon the first fundamental block named *namespace*. The objective of namespaces is to isolate each container from others and from the host kernel, by creating different views of shared resources, which include CPU, memory, storage, network. There are currently 6 namespaces [93]:

- ***MNT*** (mount points, filesystems): allows isolation of filesystem mount points seen by processes belonging to a specific container. Such separation enabled processes to view its own filesystem hierarchy;

- ***PID*** (processes): allows processes on different containers to have the same PID. The principal benefit is that during container migration all processes keep its PID number;

- ***NET*** (network stack): provides isolation of all the system resources linked with networking subsystem. Thus, a container has an *abstract network stack* with its own network interfaces, routing, firewall rules, and sockets;

- ***IPC*** (System V IPC): isolates the interprocess communication (IPC) resources (e.g., share memories, message queues);

- ***UTS*** (hostname): allows each container to have its own *hostname* and *domain name*;

- ***USER*** (UIDs): isolates the user and group ID number spaces.

Control Groups

Control Groups (cgroups) are the main mechanism behind container-based virtualization to handle shared resources usage. In fact, cgroups allow to set resource limits and access controls to specific groups of processes belonging to a particular container, in order to guarantee isolation. Cgroups mechanisms is implemented in Linux kernel by providing a tree-based hierarchical virtual filesystem. In the context of containers, cgroups is fundamental to guarantee isolation between host kernel and other containers. For example, a container must not cause severe delays (e.g., starvation phenomena) of other containers or processes on the host.

The fundamental cgroup subsystems in the Linux kernel are the following:

- ***CPU (cpu, cpuset, cpuacct)***: The CPU cgroup subsystem is used to limit a group of processes to run on specific CPUs, or for a specific amount of time;

- ***Memory***: The memory cgroup subsystem monitors memory allocation and allow to set limits for a group of processes. Furthermore, the memory subsystem controls the sum of memory and swap usage, as well as the kernel memory usage on behalf of processes;

- ***BLKIO***: Storage resources are managed through the *BLKIO* cgroup subsystem, which controls the throughput on disk by tune the read or write speeds, operations per second, queue controls, wait times and other operations on an associated block device;

- ***Devices***: The *devices* cgroup subsystem allows or denys access to devices (char or block) to group of processes;

- ***Network***: allows to control (filter) network packets based on their class (*net_ cl* subsystem) and on their priority (*net_ prio* subsystem) within processes that belonging to that groups. The control is performed to all sockets opened by processes in the group. In term of communication channel, the socket can be one spot.

Docker

Currently, Docker [49] is doubtless an hot topic in the container world, and in general in the virtualization technology panorama. As mentioned before (see 4.1.2), Docker is one of the most popular container-based virtualization technology. Docker was born in 2013 as an open source project, and during years has grown rapidly and has become a company which provides different container-based products and projects. The main differences between other container solutions, is that Docker is designed to packing and shipping individual software through building applications and all their dependencies in a easy way. Actually, Docker is not the core of container-based virtualization, instead is a management platform for containers, exploiting namespaces and cgroups to provides the basis for container operations. The Docker architecture (see Figure 4.4) consists of:

- **Docker Engine**, the core of Docker. Docker Engine is implemented via *libcontainer*[1], which in turn allows creating the required kernel namespaces, cgroups, handles capabilities and filesystem access controls to handle containers;

[1] https://github.com/docker/libcontainer

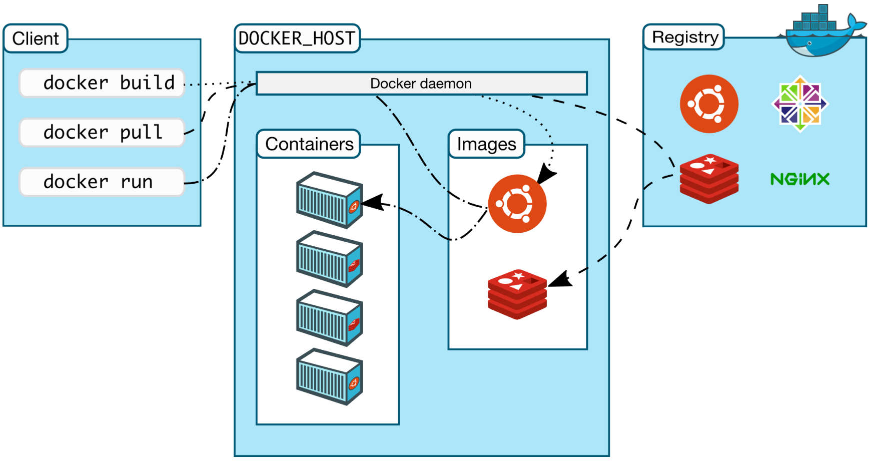


Figure 4.4. Docker architecture platform [51]

- **Docker Client**, which interacts with the *Docker daemon*, through commands (see "docker" commands). This command actually interacts the Docker daemon's REST API, using a UNIX socket;

- **Docker Daemon** accepts *Docker client* connections from the REST interface or UNIX socket and exposes Docker Engine functionality. Other tasks of docker daemon are monitoring, running, and generally acting as the "init" process for all deployed containers.

Docker containers consist of so-called *images* [50], which in turn consist of a set of read-only *layers* that represent filesystem differences. Docker allows using different storage drivers (e.g. AUFS, BTRFS, VFS, Device Mapper, OverlayFS) to stacking the layers part of a container, providing a single unified view. Each layer contains the files which have been added, changed, or deleted relative to its parent layer, starting from a base image, e.g., a version of root filesystem of an Ubuntu distribution. When a container is created, Docker adds a new readable/writable layer on top of the base image. All the read/write operations are performed on that layer, which is deleted when the container will be removed. Best practices of developing Docker services, implicitly requires that containers should be *ephemereal*, that is, the container data should not be persistent, and as soon as the container fails a new one take place with no additional configurations. The problem is that, if a container crash, any data in the read-write layer is lost. To overcome this problem, Docker introduce the *data volumes*, which allows mounting a directory on Docker host or on a shared storage, bypassing the Union File System (used by default by Docker).

Moving to the networking, Docker generally exploits network namespaces (see 4.1.2), Linux bridge, virtual ethernet pairs, and *iptables.* Docker by default create a linux bridge which allows containers to communicate each other. Each container implement network device with the virtual ethernet (*veth*), which is a Linux networking interface that allows two network namespaces to communicate. The *veth* is a full duplex link that appear a single interface in each namespace: the traffic in one interface is directed out the other interface. Docker network drivers use veths to provide explicit connections between namespaces when Docker networks are created. When a container is created and attached to a specific Docker network, one end-point of the *veth* is placed inside the container (usually seen as the *eth0* interface) while the other is attached to the Docker network. Figure 4.5 summarizes the Docker networking elements described before.

High Availability in Docker

Docker Swarm [52] is a native clustering and management system for Docker. It allows creating a cluster of Docker hosts in order to orchestrate containers. Docker Swarm guarantee high availability of containers by scheduling them across the cluster during failure conditions. In particular, Docker Swarm provides three different scheduling strategies: spread, binpack, and random. The *spread* strategy is the default used by Docker

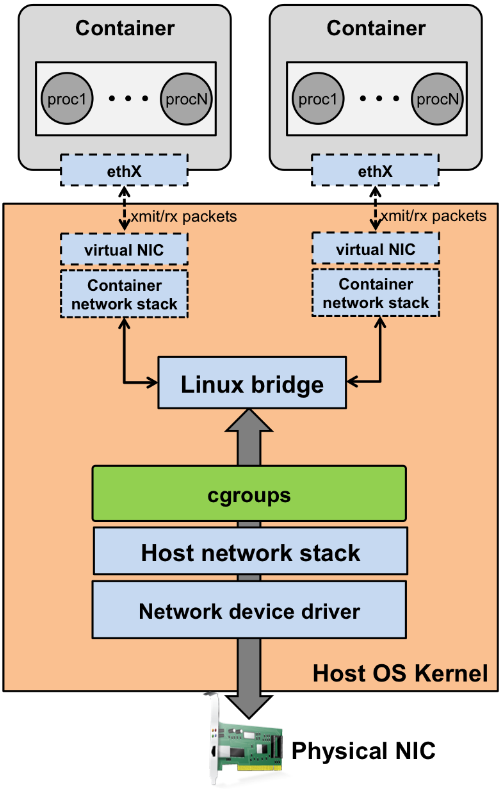


Figure 4.5. Docker networking subsystem.

Swarm. That strategy tries to place containers across all nodes in the cluster. The *binpack* strategy try to place containers on the most-loaded node, which has still resources for execution of the container being placed. In order to prevent container fragmentation, in that strategy Docker Swarm leave space for bigger containers. The *random* strategy place containers on a random node on the cluster. Furthermore, Docker Swarm allows specifying filters to schedule containers on host which complies with particular properties (e.g., host within a specific storage backend, with a specific name). The Docker Swarm architecture Figure 4.4 consists of:

- *Swarm Manager*, which manages the nodes in the cluster through Docker APIs in order to communicate with Docker Daemon (see Section 4.1.2);

- *Swarm Node*, which represents a node in the cluster. The swarm node is identified by an ID, and tracks all the information needed to the swarm manager in order to orchestrate containers. In particular, the swarm node store information about all the containers running on the host, the health state of the host, the total usage of CPU and memory;

- *Discovery Service*, which helps swarm manager nodes to adding, discovering, and managing nodes in the cluster. Docker Swarm allows using different discovery backend service, such as *etcd*, *Zookeeper*, *Consul*.

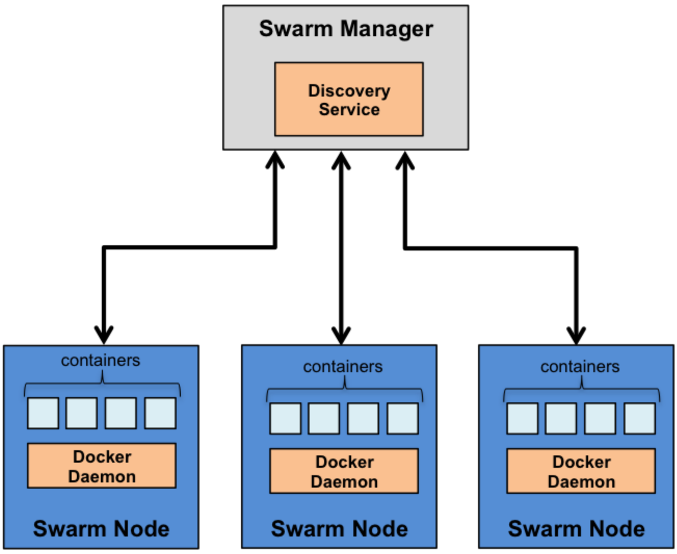


Figure 4.6. Docker Swarm architecture.

4.2 Fault Injection Tool

This section introduces a set of fault injection technologies for testing NFVIs. These technologies are based on a *fault model* that has been derived from the analysis of the typical architecture of a virtual infrastructure.

The fault model (see Section 3.3) is summarized in Figures 4.7 and 4.8. Faults can be grouped in two broad categories: faults affecting I/O subsystems (virtual/physical networks and storage), and faults affecting computation-related subsystems (virtual/physical CPUs and memory). Our analysis of fault modes pointed out that the most likely faults in virtual network infrastructures are caused by hardware faults in COTS equipment, and by software and configuration faults in the virtualization layer. These faults manifest as disruptions in I/O traffic (e.g., the transient loss or corruption of network packets, or the permanent unavailability of a network interface) and erratic behavior of the CPU and memory subsystem (in particular, corruption of instructions and data in memory and in registers, crashes of VMs and of physical nodes, and CPU and memory saturation).

These types of faults are injected by emulating their effects on the virtualization layer. In particular, I/O and Compute faults can be emulated, respectively, by deliberately injecting I/O losses, corruptions and delays (Figure 4.7), and by injecting code and data corruptions, by forcing a crash of VMs or containers, and of their hosting nodes, and by introducing CPU- and memory-bound "hogs" in the system, that is, tasks that deliberately consume CPU cycles and allocate memory areas in order to cause resource exhaustion (Figure 4.8). These faults can affect either a specific virtual node (VM or container), or a physical machine (PM) in the NFVI:

- *Guest-level faults* are focused on a specific VM or container: for instance, faults injected on network traffic from and to a specific VM or container;

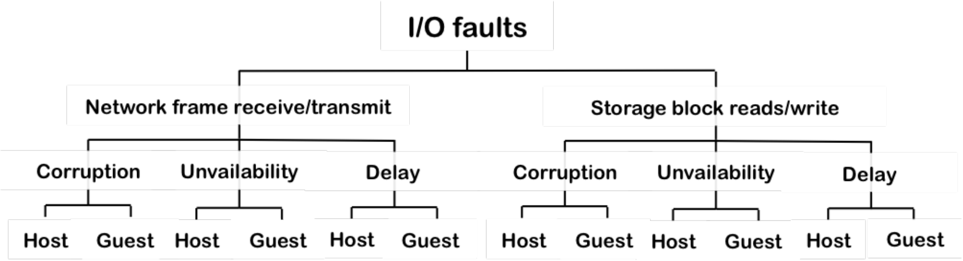


Figure 4.7. Overview of I/O faults supported by the fault injection technologies.

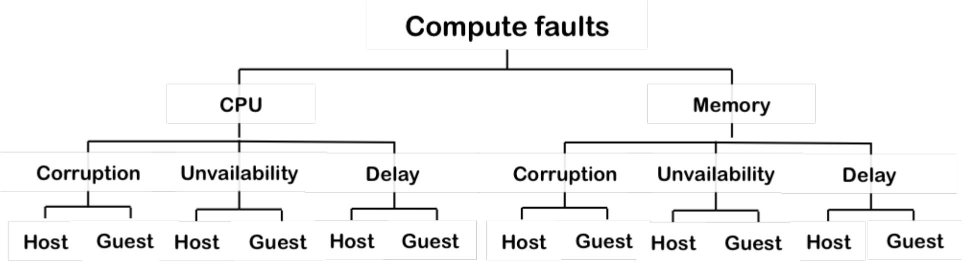


Figure 4.8. Overview of CPU and memory faults supported by the fault injection technologies.

- *Host-level faults* are focused on a physical node that hosts VMs, or physical node that hosts containers: for instance, faults injected on network traffic from and to the NFVI node, including all VMs running on the node.

These types of faults can be injected either in a *permanent* way or in a *probabilistic* way. *Permanent* faults persists in the NFVI for the whole duration of the test. This mode emulates faults that persist for a long period of time, such as unavailable hardware interfaces, or a failed upgrade or reconfiguration. To inject permanent faults, the injector drops, corrupts and/or delays all I/O transfers for the whole duration of a test (for instance, all network packets are dropped, corrupted or delayed, starting from a specific time and until the end of the test). *Probabilistic* faults affect the NFVI at random times. This mode emulates temporary faults that persist for a short period of time, such as failed I/O reads/writes due to bad disk sectors, electromagnetic interferences, worn-out connectors or partially damaged hardware interfaces. To inject probabilistic faults, the injector drops, corrupts and delays individual I/O transfers (e.g., a fault is injected one time for every ten I/O transfers).

4.2.1 Architecture

The fault injection suite include a set of tools, to be installed both on the *virtualization layer* of an NFVI node, and on its *virtual nodes*. Specifically, in the case of hypervisor-based virtualization, the virtualization layer is the hypervisor. Instead, the virtualization layer is identified by the Linux kernel in the case of container-based virtualization. Furthermore, for virtual nodes we mean Virtual Machines (VMs), in the case of hypervisor-based virtualization, and containers, in the case of container-based virtualization. Given the terminology described before, Figure 4.9 shows the high-level architecture of an NFVI node, along with fault injection tools. The tools include the following components:

- *Fault Injection Controllers.* These user space programs can be directly invoked by the user, for instance through an *ssh* session. Each of these tools *controls* the injection of a specific type of faults. For instance, one of the tools controls the injection of network faults, while another tool tools controls the injection of storage faults. In turn, these tools interact with other kernel-space tools running in the virtualization layer and in the virtual nodes;

- *Fault Injection Modules.* These programs can be both kernel modules and user space programs, and perform the actual fault injection both for the physical layer and virtual layer 3.1, implementing all the host-level faults and guest-level faults.

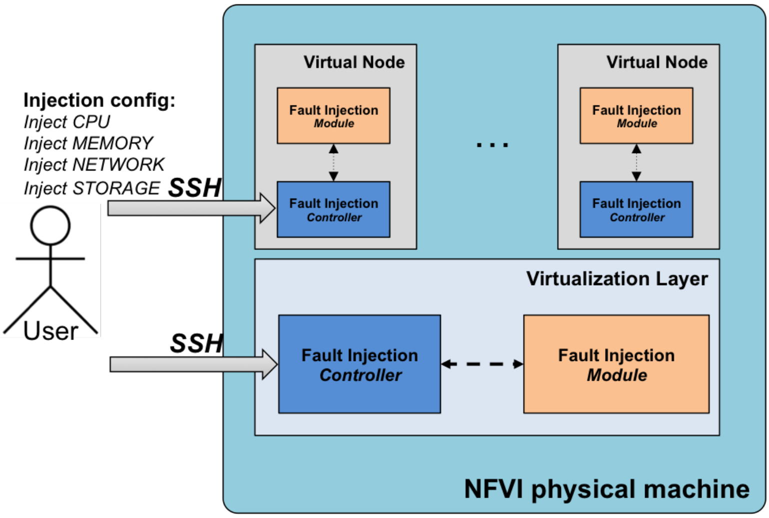


Figure 4.9. Overview of the fault injection tool architecture.

The internals of the fault injection modules for the virtualization layer and virtual node are described in the following.

4.2.2 Network Faults

Hypervisor-based scenario

In order to emulate network faults at the host level, we implement a kernel module that wraps the function responsible of sending packets over a network link. In Linux, as soon as the kernel needs to transmit a packet, it calls the *xmit()* function related to the specific network driver of the physical NIC [35]. Thus, wrapping the *xmit()* function allow us to intercept all the network traffic between physical hosts and between VMs on the NFVI. Specifically, to inject a **network unavailability** faults we drop the packet that is being transmitted by the driver to the physical NIC, by returning from the *xmit()* call. In order to inject **network delay** fault, we schedule a kernel task (specifically a workqueue [35]) that sends the

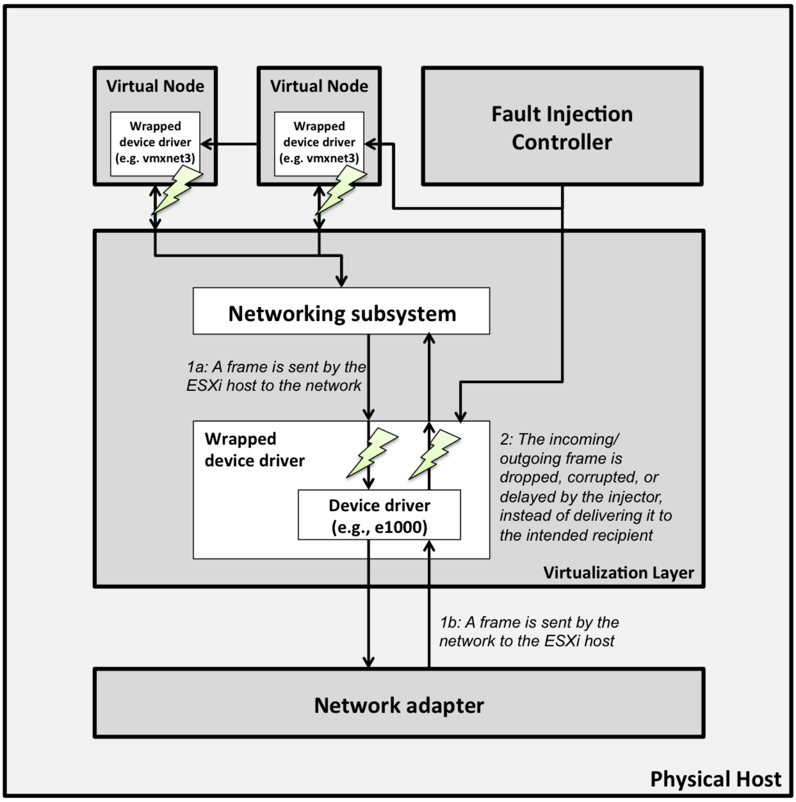

Virtual Node
Wrapped device driver (e.g. vmxnet3)
Virtual Node
Wrapped device driver (e.g. vmxnet3)
Fault Injection Controller
Networking subsystem
1a: A frame is sent by the ESXi host to the network
Wrapped device driver
Device driver (e.g., e1000)
2: The incoming/ outgoing frame is dropped, corrupted, or delayed by the injector, instead of delivering it to the intended recipient
Virtualization Layer
1b: A frame is sent by the network to the ESXi host
Network adapter
Physical Host

Figure 4.10. Internals of kernel-level tools for injecting network faults.

packet that is being to transmitted after a specified amount of delay time (e.g., after 20ms). Finally, the **network corruption** fault is implemented by obtaining the socket buffer structure (*sk_buff*), which handle every network packet in Linux kernel; once obtained the *sk_buff* belonging to the packet that is being transmitted, we corrupt the header with a specified pattern (e.g., bit-flip the first n bits). Figure 4.10 shows the internals for the network fault injector.

Actually, in order to implement guest-level faults, we exploit the same approach used for the host-level fault described above.

Container-based scenario

We exploit the same approach used in the hypervisor-based scenario by wrapping the virtual ethernet device (*veth*) (see Section 4.1.2). This approach is suitable both for host- and guest-level faults.

4.2.3 Storage Faults

Hypervisor-based scenario

In order to emulate storage faults at the host level, we implement a kernel module that wraps the functions responsible of (i) sending an I/O request commands towards the physical disk, and (ii) responsible of I/O command completion. In the Linux kernel, storage device drivers interact with the *Block I/O layer* [35] in order to serve I/O requests. In particular, each SCSI driver provides the *queuecommand()* function, which is called to enqueue a new SCSI command; moreover, as soon as the command is completed, the *done()* function is called in order to pass the results towards upper layer in the storage stack. The **storage unavailability** fault is implemented by wrapping the *queuecommand()* function and returning *DID_NO_CONNECT*[2] value as result of command completion to the SCSI upper layers (by explicitly calling the *done()* function); in this way we simulate that a physical disk is unplugged or is offline. The **storage corruption** fault is implemented by wrapping the *done()* function in order to obtain all memory mapped regions related to command that it was completed; at this time, the fault injector corrupts these regions according to the fault injection configuration. Finally, the **storage delay** fault is implemented as for the network delay fault, that is, by scheduling a kernel task that enqueue the I/O command that is being to transmitted after a specified amount of delay time (e.g., after 10000ms). Figure 4.11 shows

[2] http://www.tldp.org/HOWTO/archived/SCSI-Programming-HOWTO/SCSI-Programming-HOWTO-21.html

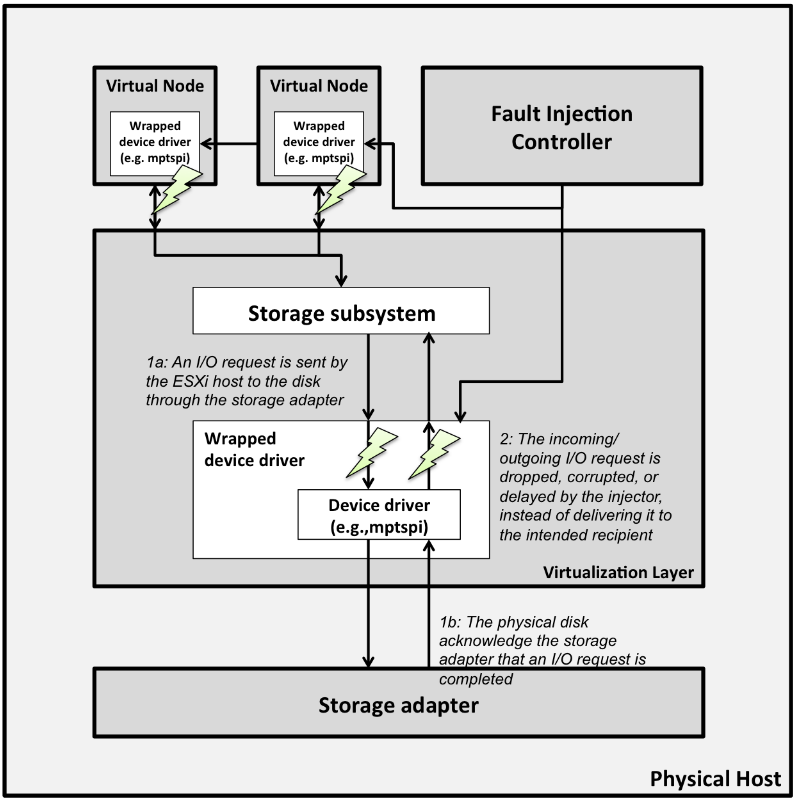

Virtual Node
Wrapped device driver (e.g. mptspi)
Virtual Node
Wrapped device driver (e.g. mptspi)
Fault Injection Controller
Storage subsystem
1a: An I/O request is sent by the ESXi host to the disk through the storage adapter
Wrapped device driver
Device driver (e.g.,mptspi)
2: The incoming/ outgoing I/O request is dropped, corrupted, or delayed by the injector, instead of delivering it to the intended recipient
Virtualization Layer
1b: The physical disk acknowledge the storage adapter that an I/O request is completed
Storage adapter
Physical Host

Figure 4.11. Internals of kernel-level tools for injecting storage faults.

the internals for the storage fault injector.

Actually, in order to implement guest-level faults, we exploit the same approach used for the host-level fault describe above.

Container-based scenario

We exploit the same approach used in the hypervisor-based scenario by wrapping the functions of the specific SCSI device which control the I/O for the shared directory of a specific container. This approach is suitable both for host- and guest-level faults.

4.2.4 CPU Faults

Hypervisor-based scenario

In this section we describe the *host-level* faults implementation, specifically the CPU faults. To emulate **CPU unavailability** fault, we simply inject a VMkernel panic to crash the ESXi host machine at specific time. In order to emulate **CPU delay** fault, we develop kernel modules that deliberately consume CPU cycles in order to cause resource exhaustion. Indeed, the kernel "hog" task interferes with the execution of user space processes, but also against other running kernel tasks, negatively impacting on the CPU utilization because the kernel gives always the highest priority to a kernel task. Since VMware ESXi is a proprietary hypervisor solution, the **CPU corruption** fault is implemented by wrapping the network device driver (actually we can choice any device driver). In fact, the driver is periodically triggered by interrupts generated from the device (i.e., the NIC), thus interrupting a VM on the host that was executing at the time of the interrupt. At this time, the wrapped device driver corrupts the current CPU state (represented by CPU registers, which are saved on a kernel stack by the interrupt handling mechanisms). The corruption of CPU registers can emulate faulty logical and arithmetical operations, bus faults, and faulty memory loads/stores, which all transit through CPU registers. Figure 4.12 shows the internals for the CPU memory corruption fault injector.

Guest-level faults bring with them some differences compared to the host-level faults. In fact, to emulate **CPU unavailability** fault we inject a kernel panic to crash the target VM at specific time. In order to emulated **CPU corruption** faults we exploit a similar approach compared to the hypervisor-based scenario. In particular, we instrument the most used system calls during operation (e.g., *open*, *read*, *write*, *mmap* system calls), in order to interrupt the current process execution, corrupting the current

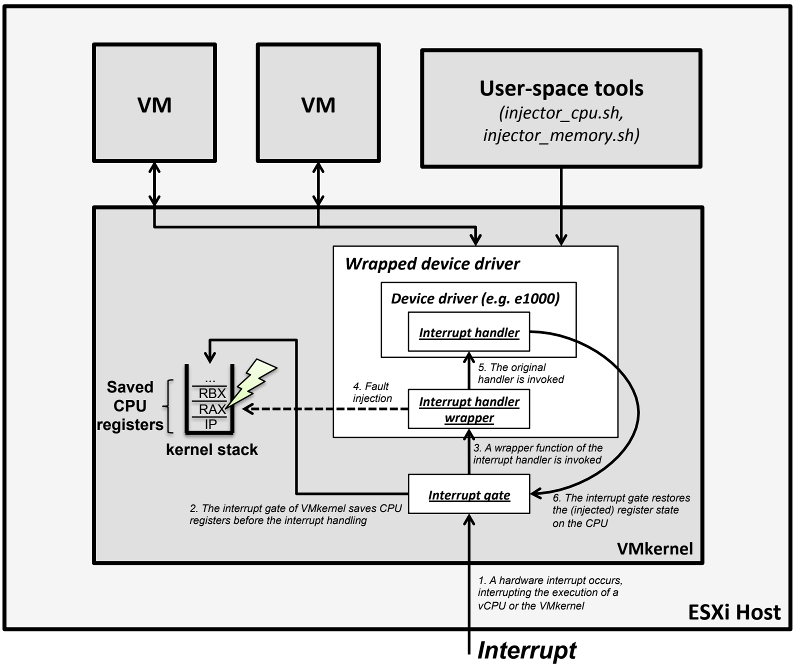

VM
VM
User-space tools
(injector_cpu.sh,
injector_memory.sh)
Wrapped device driver
Device driver (e.g. e1000)
Interrupt handler
5. The original handler is invoked
4. Fault injection
Interrupt handler wrapper
Saved CPU registers
...
RBX
RAX
IP
kernel stack
3. A wrapper function of the interrupt handler is invoked
Interrupt gate
2. The interrupt gate of VMkernel saves CPU registers before the interrupt handling
6. The interrupt gate restores the (injected) register state on the CPU
VMkernel
1. A hardware interrupt occurs, interrupting the execution of a vCPU or the VMkernel
ESXi Host
Interrupt

Figure 4.12. Internals of kernel-level tools for injecting compute faults.

CPU state. The **CPU delay** fault is emulated following the same approach of the host-level faults (see above).

Container-based scenario

In order to emulate host-level CPU faults, we exploit the same approach for the CPU *guest-level* faults used in the hypervisor-based scenario. In fact, the Docker physical host is actually a machine running a Linux kernel. Instead, for guest-level CPU faults, we slightly modify the approach. In order to emulate **CPU unavailability** fault, we simply crash the container's *init* process at specific time, by sending the SIGKILL signal[3]. In order to emulate **CPU delay**, fault we can not exploit the same approach

[3] http://man7.org/linux/man-pages/man7/signal.7.html

of host-level, because we can not execute kernel tasks within the container. To overcome the problem, we emulate CPU delay fault by running a user space program that creates (i.e., *fork*) continuously new processes that consumes CPU cycles, until the maximum number limit of running processes is reached. Also for *CPU corruption* fault we exploit a similar approach compared to the host-level faults. In particular, we instrument the most used system calls during container running, in order to interrupt its execution and corrupt the current CPU state.

4.2.5 Memory Faults

Hypervisor-based scenario

In order to emulate *host-level* memory faults, we adopt the same approach used for *host-level* CPU faults in the hypervisor-based scenario. In particular, for **memory unavailability** fault we develop a kernel modules that deliberately allocate memory areas in order to cause resource exhaustion. As before, a kernel task has higher priority rather than user processes. Thus, is more likely to generate page allocation failures, or even the activation of mechanisms to free up the allocated memory (e.g., Out Of Memory killer). The **memory corruption** fault is emulated by exploiting the CPU corruption approach. In fact, by intercepting the CPU state, we choice a register that contains a memory address; subsequently, we corrupt the first n bytes pointed by that memory address (according to overlay memory error in [144], we corrupt the first 100 bytes after the target memory address). Figure 4.12 shows the internals for the CPU and memory corruption fault injector. Finally, to emulate **memory delay** fault, we create a virtual machine such that overcommit the physical memory on the host. In fact, such VM interferes negatively with the other VMs by increasing the *page in/page out* rate, leading to an overall degradation of the memory performance.

Instead, for the *guest-level* memory faults, we implement **memory corruption** fault as for the host-level (see above). In order to implement **memory delay** fault we execute a user space program with the aim of lead to memory trashing. In particular, that program allocates new memory areas until the VM swap space is close to zero, in order to prevent the execution of OOM killer. Using this approach will substantially lower the overall memory performance. For the **memory unavailability** fault we reuse the same approach of host-level fault.

Container-based scenario

The implementation of host-level memory faults follows the same approach used in the hypervisor-based approach for the guest-level memory faults.

Instead, for guest-level memory faults, the unique difference compared to the hypervisor-based scenario for the guest-level memory unavailability, is that we can not execute kernel task within the container. Thus, we overcome that problem by intercepting the *mmap* and the *brk* system calls, which are the core system calls for the user space program memory allocation. Once we interrupt the execution of these system calls, we force to return an *ENOMEM* value, simulating that no more memory is currently available.

In the case of container-based virtualization, the fault injection approach described before can be easily used for the host-level faults, since it is the same as we injected guest-level faults. The problem is that in containers there is no virtualization of physical resources in the traditional sense(see 4.1.2), thus the approach of fault injection used in hypervisor-based virtualization for VMs is not applicable immediately. Thus, we firstly analyze how container-based virtualization abstract the physical resources, which are the target of the fault injection.

Chapter 5

Experimental Analysis of Virtualization Technologies

To better understand how to apply the dependability benchmark, and to showcase the results that can be obtained from the proposed dependability benchmark, we perform an experimental analysis on an NFV system running a virtualized *IP Multimedia Subsystem* (IMS). We deploy the IMS on two NFVIs based on different virtualization technologies: a commercial hypervisor-based virtualization platform (*VMware ESXi*), and an open-source container-based solution (*Linux* containers). The ETSI envisions the use of both hypervisor-based and container-based virtualization for NFV [119], and these two products are going to be extensively used in NFV infrastructures [157, 133, 39, 13]. Moreover, along with these two virtualization solutions, we consider two virtualization management solutions: *VMware vSphere* and *Docker*, paired respectively with VMware ESXi and Linux.

In these experiments, we adopt fault injection to analyze:

- whether degradations/outages are **more frequent or more severe** than reasonable limits;

- the impact of different types of faults, to identify **the faults to which the NFVI is most vulnerable**;
- the impact of different faulty component, to find **the components to which the NFVI is most sensitive**.

5.1 The Clearwater IMS

The VNFs running on the NFVI under evaluation are from the *Clearwater* project [31, 27], which is an open-source implementation of an IMS for cloud computing platforms. Figure 5.1 shows the components of the Clearwater IMS that are deployed on the NFVI testbed. They are:

- *Bono*: the SIP edge proxy, which provides both SIP IMS Gm and WebRTC interfaces to clients. The *Bono* node is the first point of client's connection to the Clearwater system. Clients are linked to a specific *Bono* node at their registration;
- *Sprout*: the SIP registrar and authoritative routing proxy, and handles client authentication. In Clearwater, the *Sprout* nodes form a cluster which includes a redundant memcached ([40]) cluster, which stores all the registration data and other node information;
- *Homestead*: provides an interface to *Sprout* nodes for retrieving authentication credentials and user profile information. Homestead nodes form a cluster using Cassandra [88] database;
- *Homer*: XML Document Management Server that stores MMTEL service settings for each user. The *Homer* nodes run as a cluster using Cassandra database;
- *Ralf*: a component that provides billing services.

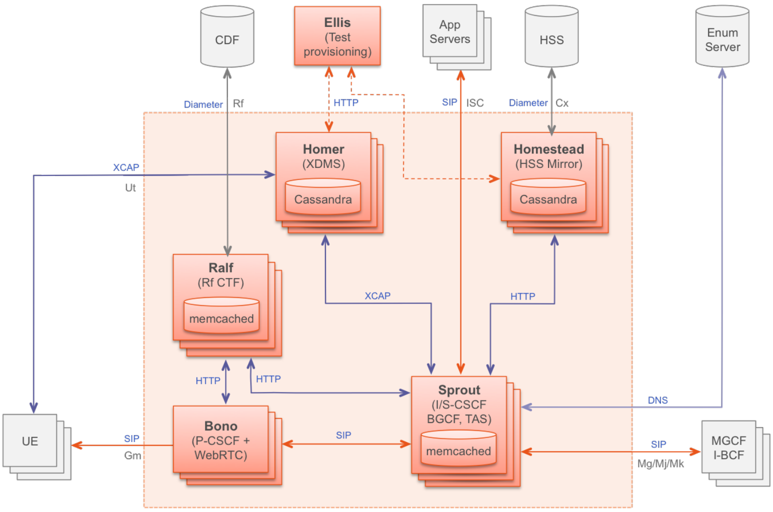


Figure 5.1. The Clearwater IMS architecture [99].

5.2 NFV Testbed

Figure 5.2 shows the testbed used for the experimental evaluation of the IMS NFV system. The same testbed has been used for experiments with both virtualization technologies, by switching between VMware ESXi with vSphere, and Linux containers with Docker Swarm, and using the same IMS VNF software in both configurations. The testbed includes:

- *Host 1* (Fault Injection Target): A machine equipped with an Intel Xeon 4-core 3.70GHz CPU and 16 GB of RAM. In the hypervisor-based scenario, this machine runs the VMware ESXi hypervisor v6.0, and hosts VMs running the VNFs of Clearwater (one VM for each VNF). In the container-based scenario, this machine runs Ubuntu Linux 14.04 OS and Docker v1.12, and hosts containers running the VNFs (one container for each VNF). Moreover, faults are injected in the resources (virtual and physical) of this host according to the fault model.

- *Host 2*: A machine with the same hardware and software configuration of Host 1. Moreover, this machine hosts active replicas of the same VNFs of Host 1, to provide redundancy to tolerate faults in the other host.

- *Tester Host*: A Linux-based computer that runs an IMS workload generator. Moreover, this machine runs a set of tools for managing the experimental workflow. These tools interact with the NFVI to deploy the VNFs, to control fault injection tools installed on the target Host 1, and to collecting performance and failure data both from the workload generator and from the other nodes of the NFVI.

- *Name, Time and Storage Server*: A machine that hosts network services (DNS, NTP) to support the execution of VNFs. Moreover, this machine hosts a shared storage service with iSCSI. The shared storage holds the persistent data of Cassandra managed by the *Homestead* and *Homer* VNFs, and the virtual disk images of the VNFs.

- *High-Availability (HA) Management Node*: A machine that runs management software: in the case of hypervisor-based virtualization, this machine runs the VMware HA service of VMware vSphere (see section 4.1.1); in the case of container-based virtualization, this machine runs the Docker Swarm master node (see 4.1.2).

- *Load Balancer*: A machine that runs a load balancer that forwards IMS requests to both Bono nodes on the two main host machines.

- A Gigabit Ethernet LAN connecting all the machines.

The workload consists of the set-up of several SIP sessions (calls) between end-users of the IMS. Each SIP session includes a sequence requests for registering, inviting other users, updating the session, and terminating the session. This workload is generated by *SIPp* [61], an open-source

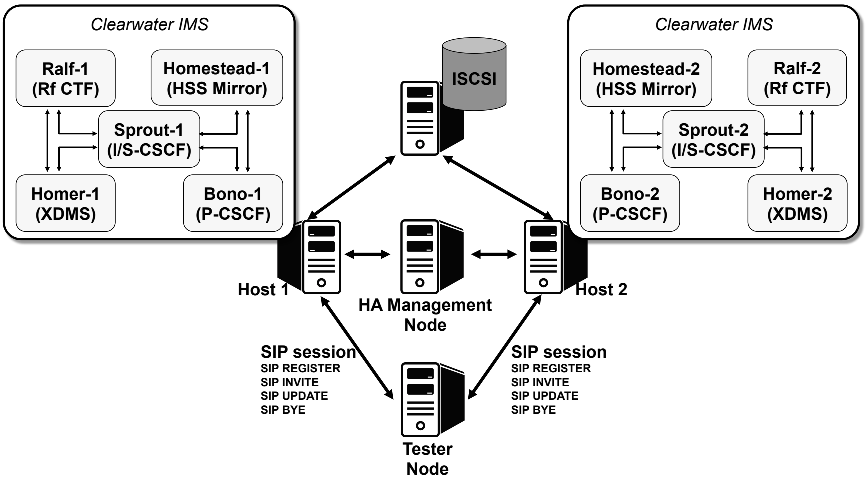


Figure 5.2. The NFVI testbed, running an IMS.

tool for load testing of SIP systems. An experiment exercises the IMS by simulating 10,000 users and 10,000 calls.

We will consider a high-availability set-up, in which each VNF is actively replicated across the hosts (Figure 5.2). The VNFs in Clearwater are designed to be stateless and to be horizontally scalable, in order to load-balance the SIP messages between the replicas using round-robin DNS. We also enable fault-tolerance capabilities provided by MANO software. In VMware vSphere, the *HA cluster* [25] capability automatically restarts a VM (in the case of virtual machine failures) or migrates it on another node (in the case of physical host failures). In the Docker configuration, we enabled *Docker Swarm* [52] to provide failure detection and automated restart for containers. To allow migration in VMware HA, we stored all VNF data (including the VM disk images) on the shared iSCSI storage; in Docker, the iSCSI storage is used to store the persistent data of *Cassandra* for the *Homer* and *Homestead* nodes.

Faults are injected in the Host 1 and its VNF replicas. In particular, we focus on injecting faults in the physical host and in the *Homestead* and

Sprout nodes, as these are the two main VNFs strictly required by the use cases of the IMS. Each experiment lasts for about 300s. We inject a fault after 80s from the start the workload, and remove the fault after 60s. As discussed in section 3.3, we consider both *I/O* and *CPU/memory* faults. Network and storage corruptions, drops, and delays are injected in the network and storage interfaces of the host, and on the virtual network and storage interfaces of *Homestead* and *Sprout*. We performed five repeated experiments for each type of fault. Overall, for each testbed configuration (hypervisor- and container-based), we perform 180 experiments (60 on the physical layer, 120 on the virtual layer), for a total of 360 experiments.

5.3 Experimental results

We computed the metrics defined by our dependability benchmark (see Section 3.2) using performance and failure data from the experiments. As basis for comparison, we computed the latency and throughput of the IMS NFV system in fault-free conditions, and compare them to the measures in faulty conditions to quantify the performance loss of the IMS. Moreover, as basis to compare the IMS unavailability, we consider that the service cannot be unavailable for more 30 seconds. This choice is an optimistic bound for the unavailability of a network service provider, as the budget can be even stricter for highly-critical services; but, as we will see in our analysis, achieving this goal using IT virtualization technologies is still a challenging problem.

Table 5.1. Fault free REGISTER requests' Latency and Throughput for VMware ESXi and Docker testbed.

Testbed	**Latency 50th %ile average [ms]**	**Latency 90th %ile average [ms]**	**Throughput [req/s]**
VMware ESXi	29.6244	57.0758	136.2439
Docker	41.5495	69.9290	135.8780

Table 5.2. Fault free INVITE requests' Latency and Throughput for VMware ESXi and Docker testbed.

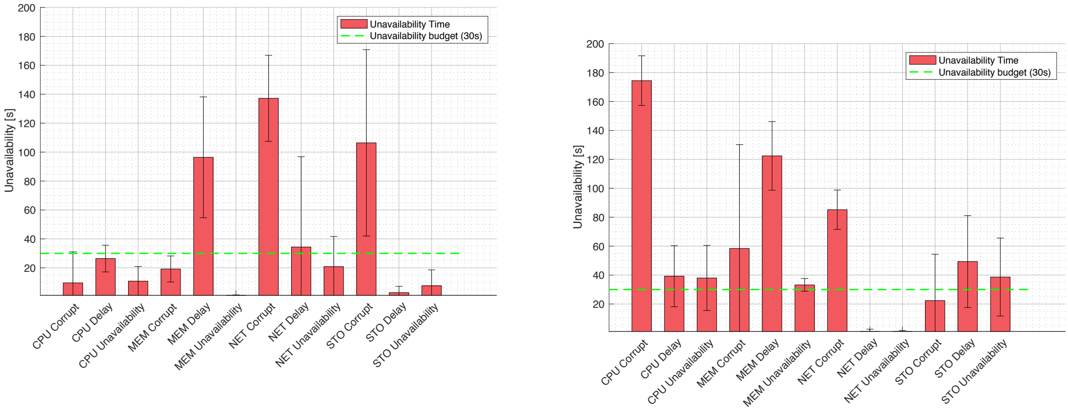
Testbed	Latency 50th %ile average [ms]	Latency 90th %ile average [ms]	Throughput [req/s]
VMware ESXi	35.43	71.34	35.8525
Docker	39.67	72.70	35.8513

The fault free analysis give to the benchmark users a basis to compare quantitatively the performance of IMS during faulty conditions The plots in the following have on x-axis each fault defined in the fault model (see Sec. 3.3), and on y-axis the value of metric computed during the fault injection experiment.

5.3.1 Service-level evaluation: Physical Layer

In this paragraph we evaluate the performance metrics related to the physical layer of NFVI, comparing VMware ESXi/vSphere and Linux/Docker scenarios. Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6 show respectively the *unavailability*, the 50th and 90th percentile of the registers' and invites' *latency*, and the registers' and invites' *throughput* computed during experiments in the VMware ESXi/vSphere and Linux/Docker.

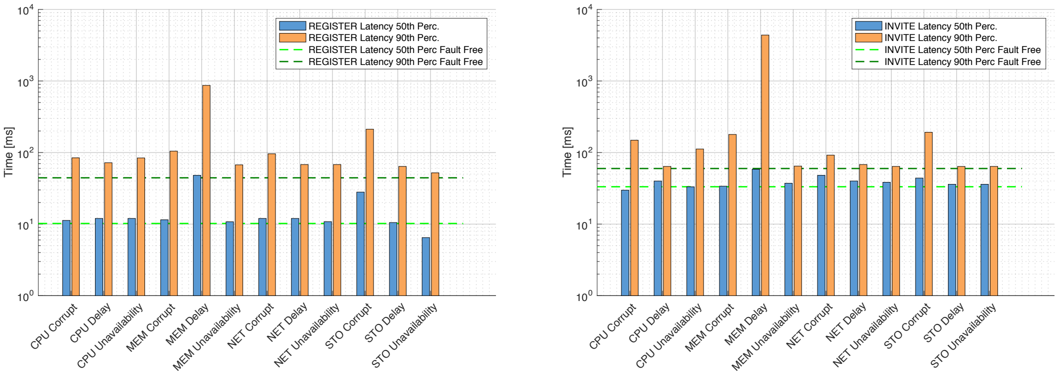
Both VMware ESXi/vSphere and Docker experiments show that all faults impact more or less on the unavailability of the system. The storage corruption fault in the VMware ESXi/vSphere scenario impact severely, by leading the service to be not recovered, both in terms of latency and throughput. In 4 out of 5 experiments, storage corruption lead ESXi host target to crash. In the other experiment, the fault lead to randomly corruption of files belonging to VMs (in one experiment corruption is about the sprout node). All the experiments trigger the failover of VMs running on failed host, i.e., restarting VMs on the healthy host. The problem was that in such cases the migration of VMs takes long time (about 5 minutes), just before the end of experiment. Of course, if we consider a



(a) VMware ESXi/vSphere

(b) Linux/Docker

Figure 5.3. VNF Unavailability under fault injection in the physical layer.



(a) REGISTER requests

(b) INVITE requests

Figure 5.4. VNF Latency under fault injection in the physical layer for VMware ESXi/vSphere.

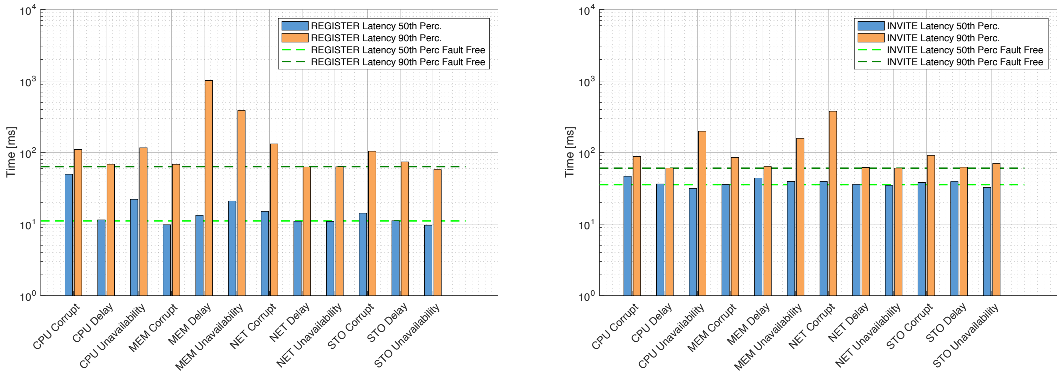
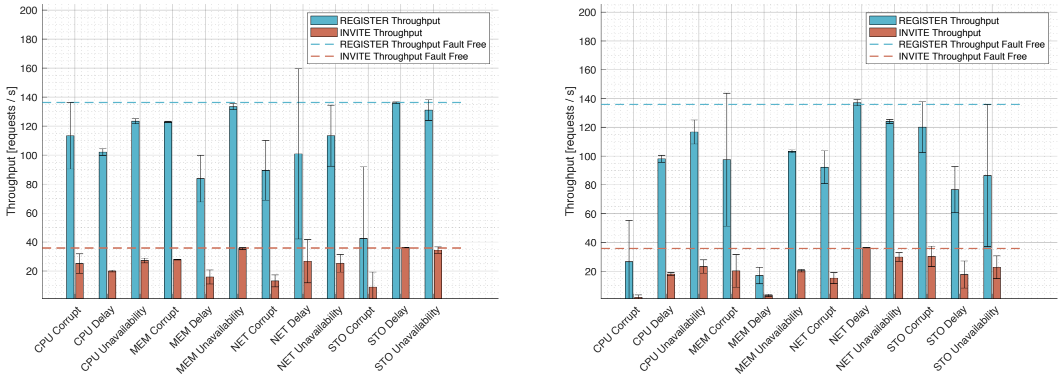


Figure 5.5. VNF Latency under fault injection in the physical layer for Linux/Docker.



(a) VMware ESXi/vSphere

(b) Linux/Docker

Figure 5.6. VNF throughput under fault injection in the physical layer.

longer experiment, the system's performance eventually will be restored, but such prolonged outage time is not acceptable in carrier-grade services. In the case of network corruption in ESXi, we can observe also an high unavailability. Such behavior is explained by the fact that ESXi heavily rely on heartbeats mechanisms to manage datastores and VMs; thus, after network corruption, the system is unable to quickly recovery.

Other type of injected faults show that in the worst case the system experiences about 100s of unavailability, and in the best cases under the unavailability budget of 30s. In particular, in the VMware ESXi/vSphere scenario, storage delay faults has a little impact on the system. This behavior is explained by the fact that VMs' virtual disks are collocated on a shared storage (iSCSI). Thus, all I/O workloads on local disk is very limited, and delay faults (10s for all I/O requests) are very well masked.

CPU faults show similar effects between VMware ESXi/vSphere and Docker testbed, except for CPU corruption faults, which does not crash the Docker host without triggering any container migration by Docker Swarm; that lead to a high unavailability. CPU unavailability fault is slightly less worse in Docker than VMware ESXi/vSphere, because of migration of containers is faster than VMs. In fact, during failover VMware ESXi/vSphere migrate and restart VMs. Instead, Docker Swarm simply restart containers, actually does not migrate anything.

Different discussion is about memory faults. Indeed, Docker scenario shows higher unavailability rather than VMware ESXi/vSphere. The behavior is what we can expect because memory isolation between virtual domains (VM and containers) and host is stronger in VMware ESXi/vSphere. In Docker, there is no virtualization layer that try to isolate perfectly guests from each other and from the host; in fact, in Docker is the Linux kernel that acts as virtualization layer, which in turn have to guarantee memory isolation through the classical OS mechanisms (pagination and segmentation). Furthermore, during memory unavailability faults, linux kernel triggers the *OOM killer*, which could kill, and subsequently force processes restart, more likely related to the workload.

Network faults impact very differently in the two scenarios. ESXi hypervisor is very complex piece of software, within a very high management network traffic volume (e.g., managing datastores liveness and locking, handling heartbeats, and so on). Furthermore, network faults in VMware ESXi/vSphere trigger failover mechanisms which impact on the unavailability of the system. On the other side, Docker exploits Swarm to handle host failover, but not network partition. Thus, network faults does not trigger failover mechanisms in Docker, with zero impact on the unavailability. Furthermore, is worth nothing that TCP/IP protocol implementation is different between ESXi and Linux.

Continuing discussion about unavailability, storage corruption fault is (with the network corruption fault) the worst for VMware ESXi/vSphere, even if all virtual disks are on shared storage. Trying to figure out why ESXi host fails, we found that storage corruptions lead the ESXi hypervisor to be unable to create new threads, and to properly handle VMFS (filesystem) operations. Concerning Docker, it shows a little amount of unavailability time due to filesystem problems, with some applications that result in segmentation fault. However, Docker host remounts filesystem in read-only, preventing host failures. Storage delay and unavailability faults have little impact on VMware ESXi/vSphere, differently from Docker scenario, in which I/O traffic on local disk is more high due to Docker subsystem processes (e.g., docker daemons).

About performance metrics, VMware ESXi/vSphere and Docker show different behaviors according to unavailability times seen before. It is worth nothing that in general the performance for VMware ESXi/vSphere, both latencies and throughput, are better than Docker in the case of CPU and memory faults. This behavior is due to the ability of hypervisor-based virtualization to guarantee stronger isolation between host and virtual machines compared to container-based virtualization. On the other side, Docker provides lower latencies and higher throughput against network faults, because at the end of fault injection it restores correctly the quality of service. Regarding storage corruption faults, according to the discussion above, VMware ESXi/vSphere provide better throughput against storage delay and storage unavailability faults.

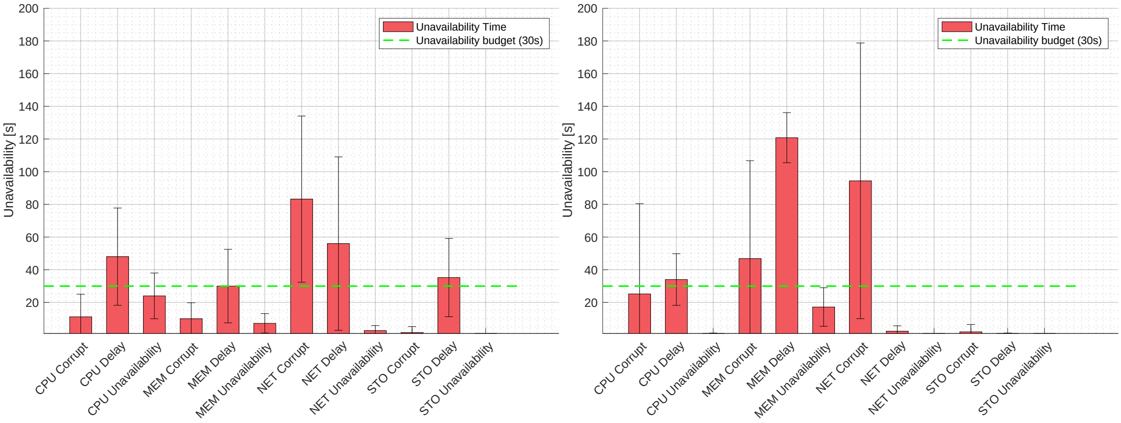
5.3.2 Service-level evaluation: Virtual Layer

In this paragraph we evaluate the performance metrics (see 3.2) related to the virtual layer of NFVI. In particular, we analyze the Homestead and the Sprout node (see 5.1), which are the most involved services in the IMS operations.

Homestead VNF

Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.10, show respectively the *unavailability*, the 50*th* and 90*th* percentile of registers' and invites' *latency*, and the registers' and invites' *throughput* computed during experiments for the VMware ESXi/vSphere and Linux/Docker testbed.

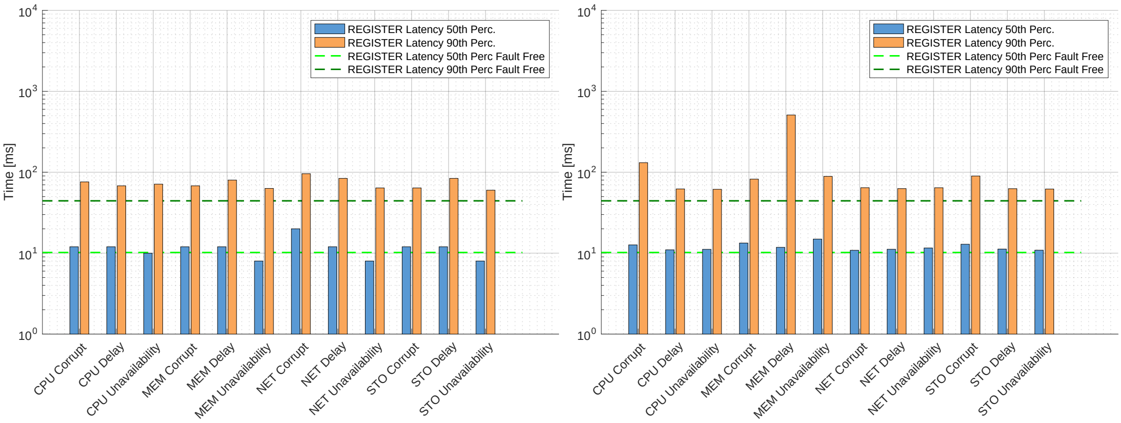
CPU faults impact significantly both on VMware ESXi/vSphere and Docker scenario, within an important amount of unavailability for CPU unavailability in the Docker scenario. In fact, such faults are emulated by crashing the VM (in the VMware ESXi/vSphere scenario) and the container (in the Docker scenario) that hosts the VNF. As soon as the crash is injected, the fault management mechanism try to failover the virtual node. In the case of VMware ESXi/vSphere, VMware HA detects the failed VM and try to restart it, within a certain amount of time. Instead, in the Docker scenario, crashing the virtual node means killing the container's *init* process. Docker has a built-in mechanism that automatic restart con-



(a) VMware ESXi/vSphere

(b) Linux/Docker

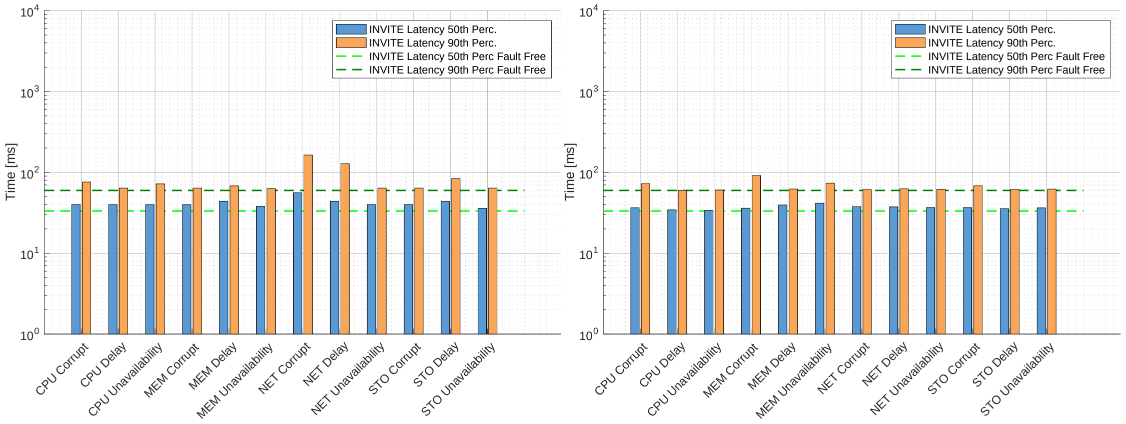
Figure 5.7. VNF Unavailability under fault injection in the *Homestead* VNF.



(a) VMware ESXi/vSphere

(b) Linux/Docker

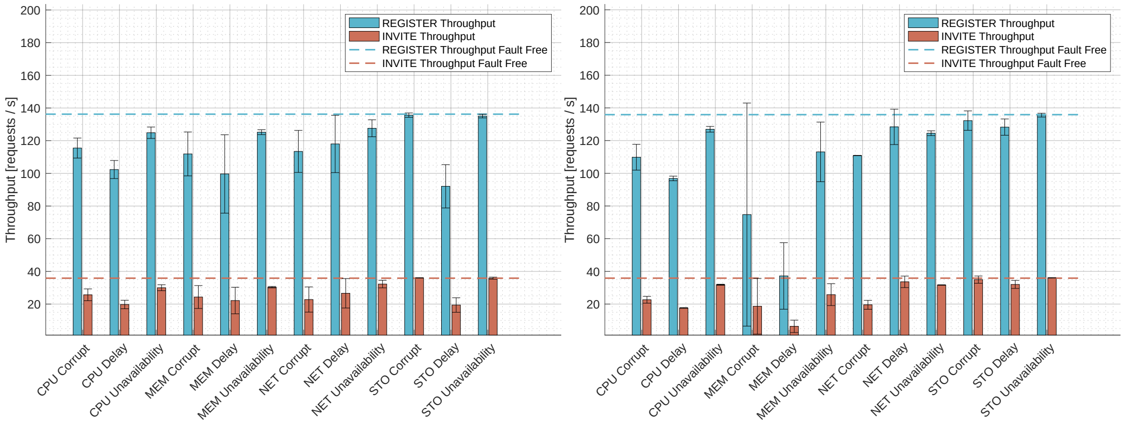
Figure 5.8. VNF latency under fault injection in the *Homestead* VNF for REGISTER requests.



(a) VMware ESXi/vSphere

(b) Linux/Docker

Figure 5.9. VNF latency under fault injection in the *Homestead* VNF for INVITE requests.



(a) VMware ESXi/vSphere

(b) Linux/Docker

Figure 5.10. VNF throughput under fault injection in the *Homestead* VNF.

tainers on exit for whatever reason, and in particular for a non-zero exit[1]. As the reader might guess, restarting a VM takes more time rather than a container. In fact, restarting a process is more lightweight rather than recreating all the hardware abstractions provided by the hypervisor and reboot a full OS. Such process takes time, also within a small VM image.

Memory faults results show clearly that ESXi isolate memory subsystem better than Docker, as physical layer results have already shown.

Network faults results show that in both VMware ESXi/vSphere and Docker scenario, the Homestead VNF is critical to deliver service properly. For network corruption faults, the VNF experience a non-negligible amount of unavailability time. Analyzing system's log, we figure out that such service unavailability is due to corruption over the TCP communication (specifically, some corruptions on the TCP window scaling process). Network delay faults behave very differently between VMware ESXi/vSphere and Docker, because the TCP/IP mechanisms are implemented differently. In fact, the VMware ESXi/vSphere experiments show higher host physical NIC drop rate due to some packets parse failures. Finally, in both VMware ESXi/vSphere and Docker scenario, network unavailability faults impact for a small amount of unavailability time (about 3s) rather network corruption faults. Such behavior is explained by the fact that once the injector does not drop packets anymore, the service performance restore almost immediately.

About storage faults at virtual layer, the results show some differences between VMware ESXi/vSphere and Docker scenario. During storage fault corruption experiments, the VMware ESXi/vSphere scenario shows low latencies because during corruptions, after a certain point, the virtual disk is mounted as read-only, differently from the case of Docker scenario. In fact, in the Docker scenario, *cassandra* detects corruptions but does not stop, leading to higher service latencies. Storage unavailability faults does not

[1] see *https://docs.docker.com/engine/admin/host_integration/* for more details.

impact on both VMware ESXi/vSphere and Docker testbed, which show the same behavior with no unavailability time and performances close to the fault free. In these cases, analyzing more deeply the log, the *cassandra* instance on the target virtual node detects generic I/O error and stop itself in order to prevent further service degradation. Storage delay faults impact on the unavailability time in Docker testbed within few failed request; on the other side, the VMware ESXi/vSphere scenario shows higher unavailability time because storage delay faults impact also on the root filesystem operations, and not only on the *Cassandra* requests.

Sprout VNF

Figure 5.11, Figure 5.12, Figure 5.13, Figure 5.14, show respectively the *Unavailability*, the 50th and 90th percentile of registers' and invites' *latency*, and the registers' and invites' *throughput* computed during experiments for the VMware ESXi/vSphere and Linux/Docker scenario.

The results show some few differences compared to the *homestead* node. Regarding CPU corruption, the processes within the *sprout* node are unable to execute instructions because the fault impact during the instruction fetch phase. On the other side, in the VMware ESXi/vSphere scenario, the corruption is detected and the VM is restarted properly.

Also considering the *sprout* node, the results confirm that memory isolation in Docker is weaker than VMware ESXi/vSphere. Indeed, the unavailability is about 150s on average. Furthermore, the discussion about differences in implementation of network stack between Docker and ESXi is also valid targeting the *sprout* node. In fact, network corruption faults impact very strongly on the unavailability of the service in the VMware ESXi/vSphere scenario rather than Docker scenario.

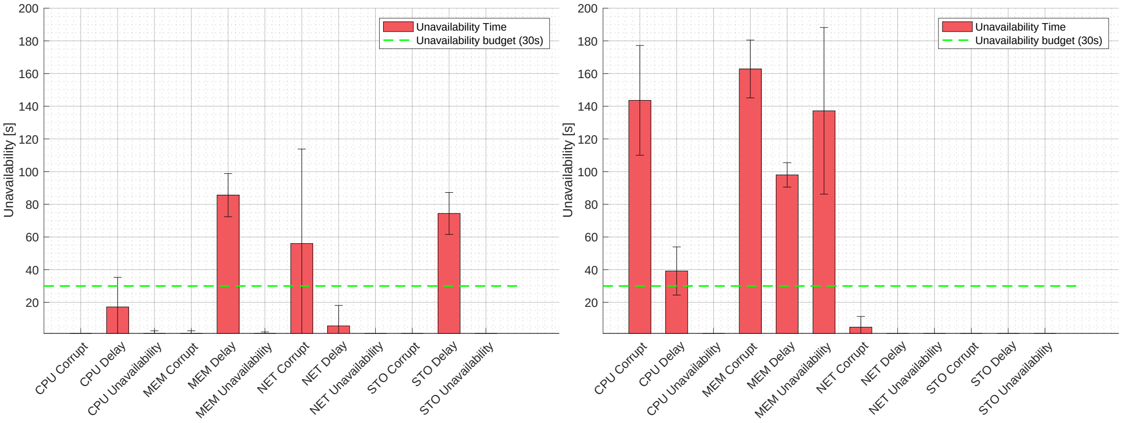
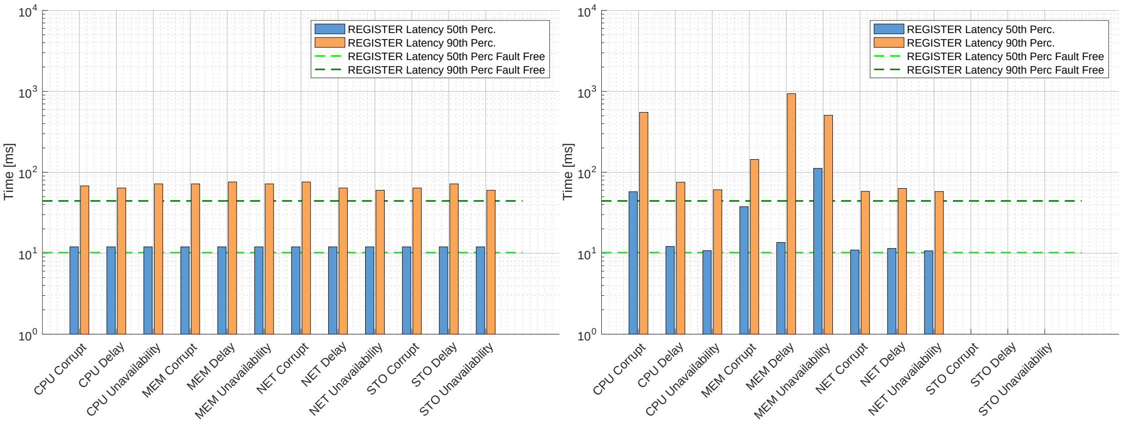


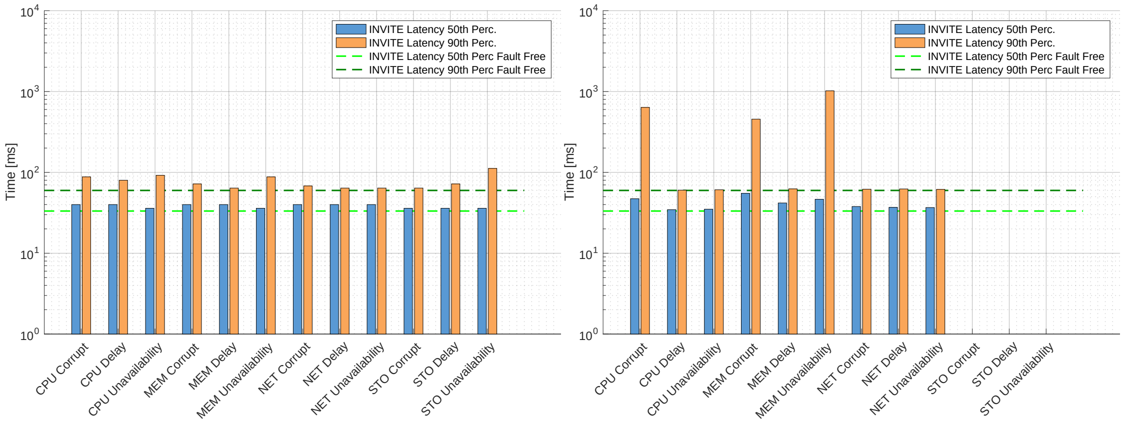
Figure 5.11. VNF Unavailability under fault injection in the *Sprout* VNF.



(a) VMware ESXi/vSphere

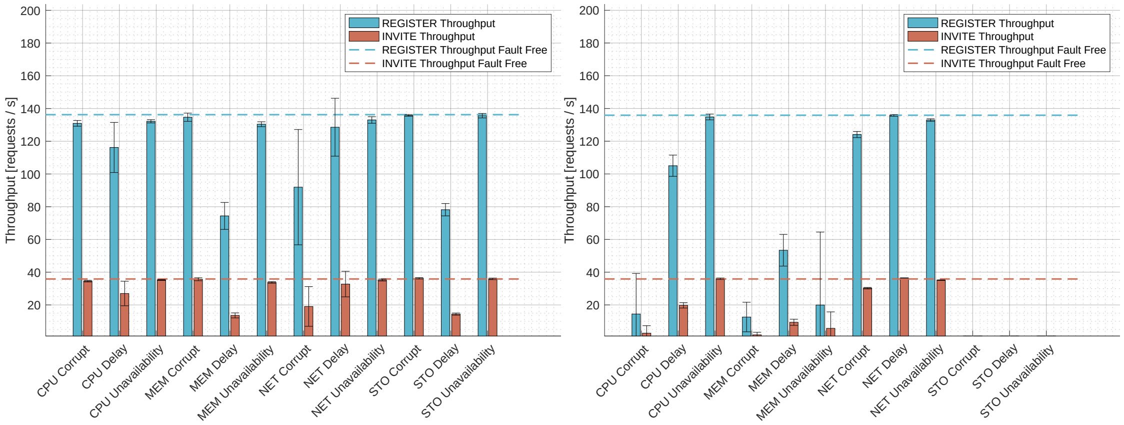
(b) Linux/Docker

Figure 5.12. VNF latency under fault injection in the *Sprout* VNF for REGISTER requests.



(a) VMware ESXi/vSphere **(b)** Linux/Docker

Figure 5.13. VNF latency under fault injection in the *Sprout* VNF for INVITE requests.



(a) VMware ESXi/vSphere **(b)** Linux/Docker

Figure 5.14. VNF throughput under fault injection in the *Sprout* VNF.

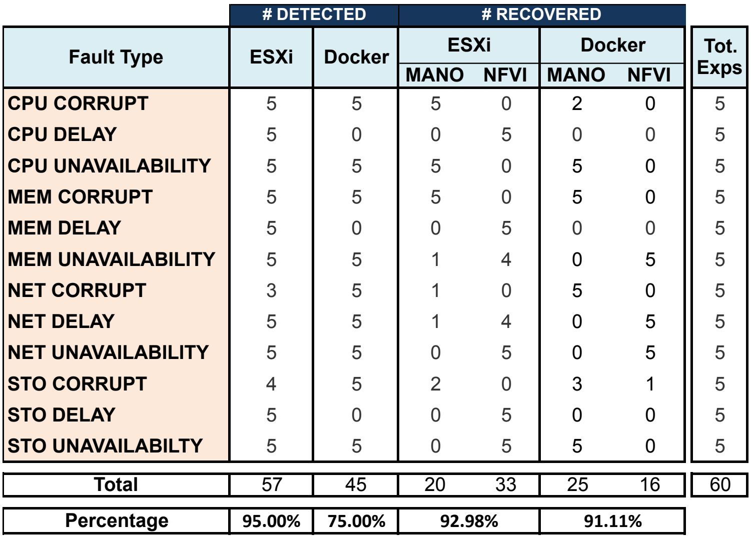
5.3.3 Infrastructure-level evaluation

We here analyze more in detail fault management in the two test configurations, by evaluating the coverage and latency of both fault detection and recovery. These measures are complementary to the service-level measures: after a fault, while the network traffic is forwarded to the healthy replicas of the VNFs, the detection and recovery are performed in background to restore the capacity of the NFV system. We use the logs collected from VMware HA and the VMkernel in the VMware ESXi/vSphere scenario, and from the Linux kernel and Docker Swarm in the other scenario. A fault is considered detected when there is at least an occurrence of log message related to fault management and to internal errors (e.g., unusual high-severity messages, and messages with specific keywords); and it is considered recovered when there is any specific message that denotes the completion of a recovery action (e.g., restart and migration) and reports that a VM or container is in a running state. The detailed results are summarized in Tables 5.3 and 5.4.

In the VMware ESXi/vSphere scenario, the fault detection coverage is about 95%, within a detection latency of 38.7s on average. For fault recovery, the coverage represents the cases where the recovery mechanisms are activated and performed successfully. The results show that in most, but not in all of the cases (storage and network corruptions), VMware ESXi/vSphere successfully performs the recovery of VMs.

In some of the experiments, the fault were tolerated or recovered locally by an NFVI node, with no interaction with the HA manager (these cases are counted in the “NFVI” column of Table 5.3, and labeled as *local recovery* in Table 5.4). This behavior was observed in the case of CPU and memory delay faults, in which VMware ESXi/vSphere detects an anomalous state during VM operations, by logging that it received intermittent heartbeats from VMs; however, after fault injection, the nodes are able to locally recover a correct service. The experiments with storage delays and

Table 5.3. Fault detection and fault recovery coverage.

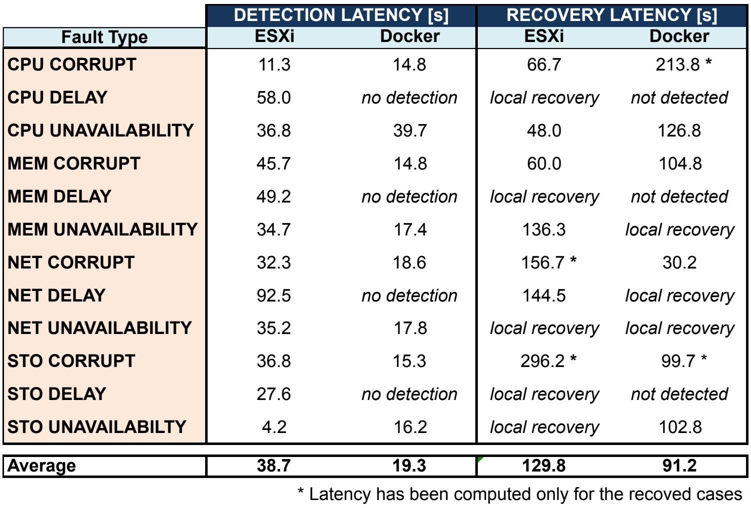


	# DETECTED		# RECOVERED				
Fault Type	ESXi	Docker	ESXi		Docker		Tot. Exps
			MANO	NFVI	MANO	NFVI	
CPU CORRUPT	5	5	5	0	2	0	5
CPU DELAY	5	0	0	5	0	0	5
CPU UNAVAILABILITY	5	5	5	0	5	0	5
MEM CORRUPT	5	5	5	0	5	0	5
MEM DELAY	5	0	0	5	0	0	5
MEM UNAVAILABILITY	5	5	1	4	0	5	5
NET CORRUPT	3	5	1	0	5	0	5
NET DELAY	5	5	1	4	0	5	5
NET UNAVAILABILITY	5	5	0	5	0	5	5
STO CORRUPT	4	5	2	0	3	1	5
STO DELAY	5	0	0	5	0	0	5
STO UNAVAILABILTY	5	5	0	5	5	0	5
Total	57	45	20	33	25	16	60
Percentage	95.00%	75.00%	92.98%		91.11%		

unavailability faults show a similar behavior. In particular, during storage delay fault experiments, the *Storage I/O Control* (SIOC) module in ESXi reports errors during the usage of the datastore; for storage unavailability faults, ESXi detects that the host datastore is inaccessible. In both cases, the datastore autonomously recovers after the injection.

Instead, the other experiments required a recovery action from the HA manager, but the recovery could not succeed in some cases. In particular, during network corruption and unavailability fault injection experiments, the injected host is unable to communicate with the other one, and VMware HA detects a *partitioned* state. In this case, VMware HA tries to migrate the VMs from the injected node to the healthy one, but it is forced to cancel the migration (as denoted by log messages such as "*CancelVm-Placement*"), due to residual data corruptions in the persistent state of VMs. In a similar way, in storage corruption faults, the migration of VMs failed to due the corruption of files with data and metadata of the VMs,

Table 5.4. Fault detection and fault recovery latency.



	DETECTION LATENCY [s]		RECOVERY LATENCY [s]	
Fault Type	**ESXi**	**Docker**	**ESXi**	**Docker**
CPU CORRUPT	11.3	14.8	66.7	213.8 *
CPU DELAY	58.0	*no detection*	*local recovery*	*not detected*
CPU UNAVAILABILITY	36.8	39.7	48.0	126.8
MEM CORRUPT	45.7	14.8	60.0	104.8
MEM DELAY	49.2	*no detection*	*local recovery*	*not detected*
MEM UNAVAILABILITY	34.7	17.4	136.3	*local recovery*
NET CORRUPT	32.3	18.6	156.7 *	30.2
NET DELAY	92.5	*no detection*	144.5	*local recovery*
NET UNAVAILABILITY	35.2	17.8	*local recovery*	*local recovery*
STO CORRUPT	36.8	15.3	296.2 *	99.7 *
STO DELAY	27.6	*no detection*	*local recovery*	*not detected*
STO UNAVAILABILTY	4.2	16.2	*local recovery*	102.8
Average	**38.7**	**19.3**	**129.8**	**91.2**

* Latency has been computed only for the recoved cases

which could not be started after the power-off. To avoid these problems, the services and protocols for fault management should be made more robust to corrupted data (e.g., by recognizing and discarding corrupted data, and retrying migration more than one time on replicated data). Moreover, since these mechanisms are provided by a third-party OTS product, it is important for the designers of the NFV system to be aware of this kind of vulnerability in the design of fault management.

Moreover, it is important to remark that the recovery latency is quite large for VMware ESXi/vSphere. The recovery process takes on average 129.8s. Part of this long time can be attributed to the policy of VMware HA that several *heartbeats* should be unanswered before declaring a node as failed (in the VMware HA terminology, the node goes from *green* to *yellow* state, and then to *red* [158]). Then, VMware HA takes a long time to restart the VMs due to the need for accessing to the shared storage, and then to allocate, initialize, and power-on the VM. Unfortunately, this process is too slow for the carrier-grade requirements of NFV.

In the Linux/Docker scenario, we observe a fault detection coverage of 75%, with a detection latency of 19.3s on average. Compared to VMware ESXi/vSphere, there is an improvement with respect to the fault detection latency, but a worse result in terms of fault detection coverage. The reason is that *Docker Swarm* uses a simpler fault detection mechanism, which monitors the network reachability with the hosts, while VMware vSphere combines both network and storage heartbeats and collects diagnostic information from the hosts. Thus, in Linux/Docker, most of the burden of fault detection is on the host on Linux kernel, which unfortunately provides little information about anomalous states (e.g., as in the case of memory overloads and other delay faults, see section 5.3.1).

However, Linux/Docker was able to recover most of the faults that were detected, as the fault recovery coverage is 91.11% with a latency of 91.2s on average. There are cases in which the fault has been detected but not recovered, such as the CPU corruption experiments: in this case, the injected host is in an anomalous state, but it is not crashed, thus Docker Swarm does not trigger the restart of the containers. In storage corruption fault injection, Docker Swarm did not migrate the containers because the fault did not impact on the host network communication, thus the host was considered alive even if the fault impacted on service availability (see also Figure 5.3b).

To summarize the results, we can state that:

- The VMware ESXi/vSphere scenario shows a **higher fault detection coverage** compared to the Linux/Docker scenario, due to more robust and ad-hoc components implemented within the fault management mechanisms provided by VMware in their products. The Docker Swarm is still to its first versions, and furthermore it implements simple mechanisms to provide high availability, and it is not able to handle performance delay faults;

- The Linux/Docker scenario shows a **lower fault recovery coverage** because for performance delay faults there is any detection. However, Linux/Docker shows a **higher fault recovery coverage** for **network** and **storage** disruptions compared to the VMware ESXi/vSphere scenario. As described in the section 5.3.1, the VMware vSphere fault management mechanisms involve a high network traffic between the node that provides fault management and the nodes being managed. Thus, network disruptions have a negative impact on the entire infrastructure, and consequently on the service deployed on it. Furthermore, in the VMware ESXi/vSphere scenario, storage corruption faults impact both on the VMs operations (i.e., corruption of VM's related files trigger the power off of the affected VMs, but in some cases they are not restarted) and on the physical host (i.e., in one experiment the host crash and some VMs are not correctly restarted on the healthy node);
- Analyzing the **fault recovery latencies**, we can observe that **VMware ESXi/vSphere outperform Linux/Docker** for the CPU corruption ($66.7s$ vs $213.8s$) and CPU unavailability ($48.0s$ vs $126.8s$) experiments. The lower latencies are due to the heartbeat requests frequency for detecting failures used by VMware HA, which is equal to $10s$ by default; instead, for Linux/Docker scenario, the Docker Engine refresh minimum interval is equal to $30s$ by default. However, the virtual layer analysis (see section 5.3.2) shows that the time needed to restart a VM is higher than restarting a Docker container.

In general, the results suggest to pay more efforts towards improving the boot time of the VM, and to increase the capacity of the nodes to speed-up the recovery process. Again, we remark that a careful fault injection experimentation is required to guide designers towards a reliable and performant NFVI. Faults in the system can have a significant impact on the availability of the system. It is not sufficient to simply provide high-availability, replicated architectures, since the occurrence of faults quickly consumes redundant resources (e.g., active replica and hosts) and reduces performance and reliability. Thus, it is advisable to adopt more complex fault tolerance strategies, by actively allocating more resources (e.g., using on-demand cloud computing resource) in the case of faults or adverse conditions, and/or by reconfiguring and recovering the failed resources.

5.4 Benchmark validation

Representativeness. As we mentioned in Sec. 2.1.1, *representativeness* deal with *measures*, the *workload* applied to the benchmark target (BT), and the *faultload* used to stimulate the fault injection target (FIT). Relating to the **measures**, we consider three general service-level measures, the throughput, the latency, and the unavailabliity, which are well-known measures used to describe quantitatively the quality of a provided network service. In addition to the service-level measures, we consider infrastructure-level measures, which are more related to the fault management. In particular we focused on the *coverage* and *latency* of the fault detection and the fault recovery mechanisms. These kind of measures are extensively used in literature in order to understand the effectiveness of the implemented fault handling mechanisms. The **workload** submitted to the SUB consists of several SIP sessions (calls) and it is very commonly used for IMS scenarios. In particular, we exploit the *sipp* workload generator that allows testers to define diverse scenarios. Different workloads and tools can be easily applied. The most difficult part of the validation for the representativeness property is about the **faultload**, and it is in general a hard task for a dependability benchmark approach. The proposed fault model (see Sec. 3.3) is based on three general types of errors, i.e., unavailability, corruption, and delay. We look for potential faults (either from software, from hardware, and from human operators) that may happen inside the target component, and that may cause errors of these three general types. To identify the faults, we studied the literature and the experience reported by practitioners about failures in the field.

Repeatability and Reproducibility. The proposed methodology consists among others of execution of a set of experiments for each defined fault. All the performed experiments are independent from each other, and the benchmark is repeatable. As outlined in the Sec. 5.2, we perform five repeated experiments for each fault, and we obtain equivalent results. Also for fault free campaigns the repeatability is guaranteed. About the reproducibility of the results, a benchmark user can exploit the proposed methodology in order to obtain the same results. Indeed, the fault injection tool is easy to configure and run on the fault injection target, the *sipp* workload generator is publicly available, and both the VMware ESXi/vSphere and Linux/Docker scenario are easy to build.

Portability. The portability is basically linked to the ability to adopt the proposed fault model for different NFVIs. In the experimental analysis, we smoothly apply successfully the proposed methodology targeting two different virtualization technologies, i.e., a commercial hypervisor-based solution, that is VMware vSphere, and an emerging container-based solution, that is, Docker.

Non-intrusiveness. The proposed fault model is applied to the FIT through the fault injection tool suite described in Sec. 4.2. The fault injection tools consists of kernel modules, and user-space scripts, and for all injected faults (CPU, memory, network, and storage) it does not require to introduce any modification neither to the benchmark target nor to the virtualization layer (i.e., the ESXi hypervisor and the Linux kernel and Docker engine).

Scalability. The proposed methodology allows benchmark users to compare systems of different size. In our case study, we inject one fault for each experiment, targeting only one component of the FIT at time . The faultload size is strictly dependent on the FIT complexity (in terms of the number of hardware and software components). In this dissertation, the FIT is the NFVI (see Sec. 3.1), which consists of hardware components and virtualization softwares. Thus, we already consider a complex target, and performing fault injection experiments were still feasible in terms of time and costs.

Chapter 6

Conclusion

6.1 Summary

Network Function Virtualization (NFV) is a new paradigm with the aim to revolutionize the network infrastructures by software implementing legacy network functions, such as load balancers, DPI, IMS. Adoption of NFV will result in dramatically reduction of the capital expenditures by replacing the expensive dedicated hardware (which constitute the main elements of traditional network infrastructures) with virtual nodes deployed on general purpose high volume servers, leveraging cloud and virtualization technologies. However, the “softwarization” process imposes software reliability concerns on future networks. While off-the-shelf hardware components are expected to fail and to be easily replaced, with very low configuration or management efforts, software (and, in particular, virtualization technologies that will be extensively used in NFV) will represent the weak point for NFV, raising new questions like: “ *What are the risks of leveraging on virtualization technologies in NFV infrastructures?*”, or “*How can we predict and mitigate the impact of faults arising from virtualization technologies?*”.

In this dissertation, we contribute to the literature by addressing the threats to dependability raised by the NFV paradigm by proposing a dependability benchmarking methodology, trying to understand if we can answer the question of which is the best virtualization technology for NFV. The proposed methodology allows understanding why the expected behaviors of a network service deployed through the NFV can be completely different from the same service deployed on legacy networks. These differences can be in terms of provided performance and reliability, which heavily depend on the complexity of fault management mechanisms, on the storage subsystems, and virtualization layer, which is subject to faults (e.g., software, and configuration faults) that are inevitable in such complex software.

The crucial aspects of the proposed methodology are both the definition and the representativeness of the fault model. In fact, the fault model should reflect all the problems the system under observation can experience in production. Moreover, the fault model should be defined by reasoning about the failure data reported in the field, or for example by analyzing the insights provided by existing experimental studies in literature. Finally, the fault model definition must consider different classes of faults (e.g., hardware, software, and configuration) which can be actually experienced by the system. However, the representativeness of the fault model is also related to its applicability in real case scenarios, because not all faults can be emulated accurately by a fault injection technique. To address these challenges, we analyze the current literature about virtualization technologies assessment, and fault injection studies for computer-based systems, in order to derive a comprehensive fault model to assess dependability in NFV. In particular, in this dissertation we propose a fault model that embrace all the components of a typical NFV infrastructure, including the CPU, memory, network, and storage resources, both at physical and virtual level.

To support and apply the methodology correctly, this dissertation presents the design and the implementation details of a fault injection tool suite. Developing fault injection in complex software systems is challenging due to the complexity of the components and their interactions. Furthermore, the black-box nature of some software solutions can be an impediment to the real applicability of the methodology. In this dissertation, we consider the two most popular virtualization approaches in use in the NFV context, and in general in the cloud. In particular, we target a commercial hypervisor-based solution, that is, VMware vSphere, and an emerging container-based solution, that is, Docker. The fault injection tools allow putting into the targeted virtualization technologies the desired exceptional conditions (e.g., CPU corruptions, memory unavailability, network and storage disruptions), in order to observe the consequent behavior, and assessing the performance and dependability of the system by measuring metrics about quality of service and effectiveness of fault management mechanisms.

6.2 Analysis of Results

The experimental results show clearly that performance and reliability are critical objectives for the widespread adoption of NFVIs. The case study on the IMS showed how the proposed methodology can point out dependability bottlenecks in the NFVI and guide design efforts. In particular, the results highlight both the strengths and the weaknesses between the VMware vSphere and Docker. In the following, we summarize the main insights provided by the experimental results.

Service-level evaluation, faults in the physical layer. The analysis of the *physical layer* shows that VMware ESXi/vSphere is in general more robust against memory faults compared to Linux/Docker, especially for memory corruption and memory unavailability faults, which lead the system to an unavailability time of around $58.4s$ and $33.2s$ respectively. That behavior is what we would expect because in Docker there is no virtualization layer (in the sense of hypervisor-based solutions) that isolate guests from each other and guest from host; thus, the memory isolation in such cases is not perfectly guaranteed. These faults also impact severely on service latencies. On the other hand, network disruptions and storage corruption heavily impact on the performance and dependability of the service deployed on VMware vSphere, which show in the worst case (network corruption fault) an unavailability time of $137s$ on average, which is unacceptable for carrier-grade service requirement;

Service-level evaluation, faults in the virtual layer. The analysis of the *virtual layer* shows again some differences between VMware ESXi/vSphere and Linux/Docker. Targeting the *Homestead* node, the results show that CPU faults impact on the service unavailability on both scenarios. It is worth noting that the lower unavailability time against CPU unavailability (i.e., injection of virtual node crash) for Linux/Docker scenario (around $0.33s$) suggest that the process of restarting a container is more lightweight rather than a VM, as we would expect. Furthermore, the results reveal also that the different implementations of the network and storage subsystem for the two virtualization technology produce distinct performance results. In particular, in the VMware ESXi/vSphere scenario network delays lead the service to experience a high unavailability time due to a consequent high packet drop rate; on the other side, the same kind of fault do not seems to have much impact on service deployed on Linux/Docker. Finally, in the VMware ESXi/vSphere, storage delays impact on the service unavailability because, unlike the Linux/Docker scenario, also the root filesystem is subject to the injected faults. Regarding the analysis of virtual layer by targeting the *sprout* node, the results confirm that *memory isolation* in Docker is weaker than ESXi. Furthermore, the discussion above about network delay faults for the *Homestead* node, is valid for the network corruption faults in the case of VMware ESXi/vSphere scenario. Finally, CPU corruption faults seems to be unrecoverable in the Docker scenario, when such fault does not crash the host machine (this is the case for the *Sprout* node);

Infrastructure-level evaluation. The analysis of the fault management mechanism provided by the targeted virtualization solutions shows that VMware ESXi/vSphere provides a higher fault detection coverage (about 95%) rather than Docker (about 75%). This results is due to the ad-hoc solutions provided by VMware for fault management, which is more tested and robust. Furthermore, fault recovery measures highlights that the triggered failover actions not always perform a successful recovery, both in VMware ESXi/vSphere and Linux/Docker. In fact, the fault recovery coverage in VMware ESXi/vSphere and Linux/Docker is respectively 92.98% and 91.11%. In the VMware ESXi/vSphere scenario, by analyzing more deeply the fault management logs, we figure out that in some experiments the migration of VMs is not successful, or even if the migration is correctly performed, some VMs are not restarted on the healthy host due to residual effects of injected faults. Instead, for Docker scenario, in some experiments we observe that the *swarm manager* node is not able to detect a dead Docker host, or it is unable to mount correctly the partition on the shared storage for specific containers. About the fault recovery latencies, for the VMware ESXi/vSphere and Linux/Docker scenario, we observe respectively 129.8s and 91.2s on average to recovery from faults. However, analyzing the common faults in which are required failover actions, ESXi outperform Docker to recover from CPU corruption and CPU unavailability faults (i.e., 66.7s vs 213.8s, and 48.0s vs 126.8s

respectively), while Docker takes less time rather than VMware vSphere to recover from network corruption and storage corruption faults (i.e., 30.2s vs 156.7s, and 99.7s vs 296.3s respectively). Such differences highlight that the *heartbeat* mechanism has a significant influence on the recovery latency, thus experimenting with different configurations of the heartbeat period could allow designers to achieve a more efficient recovery. Moreover, the results suggest that designers should optimize the time-to-reboot of VMs to speed-up the recovery process.

6.3 Discussion

Best virtualization technologies for carrier-grade networks. To answer the question of which is the best virtualization technology for NFV we need to understand thoroughly how these technologies (including both the virtualization solutions and the provided fault management mechanisms) behave under faulty conditions. In general, the carrier-grade requirement should be ensured whatever virtualization is intended to be used by NFV architects. The discussion above underlines that for some faults categories the VMware vSphere solution guarantee strongest isolation for memory, but higher unavailability time against network and storage disruptions rather than the Docker solution. Furthermore, Docker seems to provide lower recovery latencies for restarting a specific virtual node due to the lightweight nature of container-based solution. Conversely, the provided fault detection coverage in Docker is higher than VMware vSphere, which provides more robust and tested fault management mechanisms (i.e. VMware HA). Therefore, an NFV developer must consider the trade-offs seen before between the adoption of different virtualization technologies in order to be guided for designing of NFVI and developing of countermeasures for detecting faults;

Virtualization adds new threats to network softwarization. Leveraging virtualization to deploy network services through the NFV paradigm introduce new challenges that need to be addressed. In particular, the expected behaviors in NFV can be completely different from the same service provided by deploying it on legacy networks. Such differences are in terms of provided performance and reliability due to the complexity of fault management mechanisms, dependence on storage subsystem, and virtualization layer faults;

Fault injection for NFV assessment. Fault injection proves to be valuable approach to introduce uncommon scenarios in the NFVI, which can be fundamental to provide a high reliable service in production. Developing fault injection tools is hard in complex software systems, especially within proprietary solution (e.g., the targeted VMware vSphere) and it needs to be carefully planned in the context of NFV. Developers can easily reuse the proposed methodology to compare two different VNF products (e.g., two alternative IMS products) using the same NFVI and virtualization technology; or evaluate two different physical setups (e.g., by only varying the number and type of hardware machines); or compare different MANO products using the same NFVI. The purpose of this work has been to provide a general and flexible methodology that could be used for benchmarking different configurations. Moreover, it could be possible to further extend the methodology to evaluate other highly-critical cloud-based services beyond NFV.

6.4 Future directions

Other technologies and approaches for virtualization. In this dissertation, we analyze two different virtualization approaches, i.e., hypervisor-based and container-based. However, the technology solutions that can be adopted to implement a NFV infrastructures can be various, both from hypervisor-based approaches (e.g., kvm, XEN) and from container-based approaches (e.g., LXD, Kubernetes). Furthermore, aside from virtualization technologies, in last years different approaches to the virtualization come up. For example, *unikernels* are extremely optimized and highly performant kernels (e.g., ClickOS, Mirage OS, OSv) that allows running a single application on a single VM. That approach rely on the underlying hypervisor to guarantee isolation. The proposed methodology can be applied considering this panorama of virtualization technologies in order to pin point their strengths and weakness.

NFV acceleration technologies. The NFV infrastructure includes almost all hardware and software components to provided the network service through the VNFs. However, in order to meet the stringent performance requirements (e.g., service latency), the VNFs need some form of network *acceleration.* Such technologies will be largely used in NFV deployments, and can be performed both in hardware, with specialized devices (e.g., 6Wind virtual accelerator) and in software, with specific frameworks (e.g., Intel DPDK, OpenDataPlane); however, the accelerators can be implemented also in an hybrid form. The proposed methodology can ben applied in scenarios where these accelerator are implemented.

Software Defined Networking (SDN). Software Defined Networking (SDN) is an emerging network architecture where network control is decoupled from data forwarding, and which is directly programmable. In SDN, a (logically) centralized controller sends commands to the devices about how to route the packets on the network. The controller is responsible for maintaining all of the network paths, as well as programming each of the network devices it controls. This allows a more reliable and efficient network. The ETSI has clarified that the NFV paradigm is independent from SDN, and that the combination of these technologies is straightforward. NFV can be used on top of SDN. A key point in the softwarization process of networks could be apply the proposed methodology to the SDN context, targeting either a pure SDN or a SDN/NFV scenario.

NFV benchmarking in production. In our case study, we apply the proposed methodology by targeting a NFV testbed made in laboratory. The problem is that, network operators have to guarantee that the provided network service still tolerate failures in *production*, where there are different conditions that can not be reproduced precisely in a controlled environment. In particular, we can leverage fault injection to introduce uncommon scenarios also in production environment. Thus, we should understand how the proposed methodology can be applied for NFV production scenarios.

This page intentionally left blank.

Bibliography

[1] ISO/IEC 25010:2011, Systems and software Quality Requirements and Evaluation (SQuaRE), author=ISO/IEC, year=2011.

[2] Cloud computing for financial markets. White Paper, 2011.

[3] Your cloud in healthcare. White Paper, 2011.

[4] Network Functions Virtualisation (NFV) - Network Operator Perspectives on Industry Progress. White Paper, 2013.

[5] Shehla Abbas, Mohamed Mosbah, Akka Zemmari, and Université Bordeaux. Itu-t recommendation g.114, “one way transmission time. Technical report, 2003.

[6] Advanced Micro Devices, Inc. . Virtualization Solutions. http://www.amd.com/en-gb/solutions/servers/virtualization.

[7] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *ACM SIGOPS Operating Systems Review*, 44(4):3–18, 2010.

[8] A. Albinet, J. Arlat, and J.C. Fabre. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. Intl. Conf. on Dependable Systems and Networks*, 2004.

[9] Amazon. Amazon EC2 Container Service.

[10] Amazon, Inc. Linux AMI Virtualization Types. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/virtualization_types.html.

[11] Amazon.com, Inc. Summary of the amazon ec2 and amazon rds service disruption in the us east region, April 2011.

[12] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual cpu validation. In *Proc. SOSP*, 2015.

[13] Jason Anderson, Hongxin Hu, Udit Agarwal, Craig Lowery, Hongda Li, and Amy Apon. Performance considerations of network functions virtualization using containers. In *Proc. ICNC*, 2016.

[14] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Trans. on Software Engineering*, 16(2), 1990.

[15] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1), 2004.

[16] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 631–636. IEEE, May 2010.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. SOSP*, pages 164–177, 2003.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, volume 37, page 164, New York, New York, USA, October 2003. ACM Press.

[19] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments using FIAT. *IEEE Trans. on Comp.*, 39(4), 1990.

[20] Eric Bauer and Randee Adams. *Reliability and Availability of Cloud Computing.* Wiley-IEEE Press, 1st edition, 2012.

[21] Alysson Bessani, Rüdiger Kapitza, Dana Petcu, Paolo Romano, Spyridon V. Gogouvitis, Dimosthenis Kyriazis, and Roberto G. Cascella. A look to the old-world sky: EU-funded dependability cloud computing research. *SIGOPS Operating Systems Review*, 46(2):43–56, July 2012.

[22] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *Proc. DBTest*, 2009.

[23] Aaron Brown. Towards availability and maintainability benchmarks: A case study of software raid systems. Technical report, DTIC Document, 2001.

[24] Aaron B Brown, Leonard C Chung, and David A Patterson. Including the human factor in dependability benchmarks. In *Workshop on Dependability Benchmarking*, 2002.

[25] Mike Brown, Anil Kapur, and Justin King. VMware vCenter Server 5.5 Availability Guide. Technical report, 2014.

[26] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[27] Giuseppe Carella, Marius Corici, Paolo Crosta, Paolo Comi, Thomas Michael Bohnert, Andreea Ancuta Corici, Dragos Vingarzan, and Thomas Magedanz. Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures. In *Proc. ISCC*, 2014.

[28] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *Proc. Intl. Conf. on Dependable Computing for Critical Applications*, 1995.

[29] F. Cerveira, R. Barbosa, and H. Madeira. Soft errors susceptibility of virtualization servers. In *Proc. PRDC*, 2017.

[30] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults based on Field Data. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, 1996.

[31] Clearwater. Project Clearwater - IMS in the Cloud, 2014.

[32] Cloud Watch HUB. Cloud certification guidelines and recommendations.

[33] Cloudera. Cloudera Homepage. http://www.cloudera.com/.

[34] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 143, New York, New York, USA, June 2010. ACM Press.

[35] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[36] Domenico Cotroneo, L De Simone, AK Iannillo, Anna Lanzaro, Roberto Natella, Jiang Fan, and Wang Ping. Network function virtualization: Challenges and directions for reliability assurance. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 37–42. IEEE, 2014.

[37] Domenico Cotroneo, Luigi De Simone, Antonio Ken Iannillo, Anna Lanzaro, and Roberto Natella. Dependability evaluation and benchmarking of network function virtualization infrastructures. In *Proc. NetSoft*, 2015.

[38] F. Cristian. Exception handling and software fault tolerance. *IEEE Trans. on Computers*, C-31(6), 1982.

[39] Richard Cziva, Simon Jouet, Kyle JS White, and Dimitrios P Pezaros. Container-based network function virtualization for software-defined networks. In *Proc. ISCC*, 2015.

[40] Danga Interactive. Memcached Home Page. https://memcached.org/.

[41] Antonio Dasilva, José-F Martínez, Lourdes López, Ana-B García, and Luis Redondo. Exhaustif®: A fault injection tool for distributed heterogeneous embedded systems. In *Proc. EATIS*, 2007.

[42] David Linthicum. Calculating the true cost of cloud outages.

[43] DBench Project. Fault Representativeness. http://webhost.laas.fr/TSF/DBench/Deliverables/ETIE2.pdf, 2002.

[44] DBench project. *DBench Final Report*. http://www.laas.fr/DBench/, 2004.

[45] DBench Project. Dependability Benchmarking Concepts. http://webhost.laas.fr/TSF/DBench/Final/DBench-ch1.pdf, 2004.

[46] Pradipta De, Anindya Neogi, and Tzi-cker Chiueh. Virtualwire: A fault injection and analysis tool for network protocols. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 214–221, 2003.

[47] Luigi De Simone. Towards fault propagation analysis in cloud computing ecosystems. In *Proc. ISSREW*, 2014.

[48] Larry Dignan. Amazon explains its S3 outage. http://www.zdnet.com/blog/btl/amazon-explains-its-s3-outage/8010.

[49] Docker Inc. Docker HomePage.

[50] Docker, Inc. Docker Image Specification v1.0.0. https://github.com/docker/docker/blob/master/image/spec/v1.md.

[51] Docker, Inc. Docker Overview. https://docs.docker.com/engine/understanding-docker/.

[52] Docker Inc. Docker Swarm. https://www.docker.com/products/docker-swarm.

[53] J. Durães and H. Madeira. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. Pacific Rim Intl. Symp. on Dependable Computing*, 2002.

[54] J. Durães and H. Madeira. Generic Faultloads based on Software Faults for Dependability Benchmarking. In *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*, pages 285–294, 2004.

[55] J. Durães, M. Vieira, and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior. *IEICE Trans. on Inf. and Sys.*, 86(12):2563–2570, 2003.

[56] Eclipse Foundation. The AspectJ Project. http://eclipse.org/aspectj/.

[57] European Union Agency for Network and Information Security. Cloud computing certification.

[58] Jesús Friginal, David de Andrés, J-C Ruiz, and Regina Moraes. Using Dependability Benchmarks to Support ISO/IEC SQuaRE. In *Proc. PRDC*, pages 28–37, 2011.

[59] Hajime Fujita, Yutaka Matsuno, Toshihiro Hanawa, Mitsuhisa Sato, Shinpei Kato, and Yutaka Ishikawa. DS-Bench Toolset: Tools for dependability benchmarking with simulation and assurance. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–8. IEEE, June 2012.

[60] Franco Fummi, Davide Quaglia, and Francesco Stefanni. Network fault model for dependability assessment of networked embedded systems. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*, pages 54–62. IEEE, 2008.

[61] Gayraud, Richard and Jacques, Olivier and Day, Robert and Wright, Charles P. SIPp. http://sipp.sourceforge.net/.

[62] Gigaom.com. Windows azure outage hits europe, July 2012.

[63] Google Inc. Google Cloud Platform.

[64] Google, Inc. Google Compute Engine - IaaS | Google Cloud Platform. https://cloud.google.com/compute/.

[65] Google Inc. Kubernetes.

[66] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Proc. of SRDS*, 1985.

[67] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: a framework for cloud recovery testing. page 18, March 2011.

[68] Hadoop. Hadoop Homepage. http://hadoop.apache.org/.

[69] S. Han, KG Shin, and HA Rosenberg. DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment. In *Proc. CPDS*, 1995.

[70] HP Networking Performance Team. Netperf HomePage. `http://www.netperf.org/netperf/`.

[71] M.C. Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4), 1997.

[72] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[73] JJ Hudak, B.H. Suh, DP Siewiorek, and Z. Segall. Evaluation and Comparison of Fault-Tolerant Software Techniques. *IEEE Trans. on Reliability*, 42(2), 1993.

[74] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. page 11, June 2010.

[75] ISO. Product development: software level. *ISO 26262: Road vehicles – Functional safety*, 6, 2011.

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

[77] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11*, volume 46, page 171, New York, New York, USA, October 2011. ACM Press.

[78] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: a programmable tool for multiple-failure injection. *ACM SIGPLAN Notices*, 46(10), 2011.

[79] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On fault resilience of OpenStack. In *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13*, pages 1–16, New York, New York, USA, October 2013. ACM Press.

[80] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[81] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proc. FTCS*.

[82] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[83] W.-I. Kao and R.K. Iyer. DEFINE: A Distributed Fault Injection and Monitoring Environment. In *Proc. Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994.

[84] W.-I. Kao, R.K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE TSE*, 19(11), 1993.

[85] J. Katcher. Postmark: A New File System Benchmark. Technical Report TR-3022, 1997.

[86] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[87] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Trans. on Software Engineering*, 26(9), 2000.

[88] Avinash Lakshman and Prashant Malik. Cassandra. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010.

[89] Laprie, J.-C. From dependability to resilience. In *Proc. Intl. Conf. DSN, Supplemental Volume, Fast Abstracts*, 2008.

[90] Michael Le and Yuval Tamir. Rehype: Enabling vm survival across hypervisor failures. In *ACM SIGPLAN Notices*, volume 46, pages 63–74, 2011.

[91] Michael Le and Yuval Tamir. Fault injection in virtualized systems—challenges and applications. *Dependable and Secure Computing, IEEE Transactions on*, 12(3):284–297, 2015.

[92] Zhenjiang Li, Cheng Chen, and Kai Wang. Cloud computing for agent-based urban transportation systems. *Intelligent Systems, IEEE*, 26(1):73–79, Jan 2011.

[93] Linux man-@pages project. NAMESPACES(7). Linux Programmer's Manual. http://man7.org/linux/man-@pages/man7/namespaces.7.html.

[94] LWN.net. Namespaces in operation, part 1: namespaces overview.

[95] LXC. LXC - Linux Containers.

[96] Antonio Manzalini, Roberto Minerva, E Kaempfer, F Callegari, Aldo Campi, Walter Cerroni, Noël Crespi, E Dekel, Y Tock, Wouter Tavernier, et al. Manifesto of edge ICT fabric. In *Proc. ICIN*, pages 9–15, 2013.

[97] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, NIST, Gaithersburg, MD, United States, 2011.

[98] Paul Menage. CGROUPS.

[99] Metaswitch Networks. Clearwater Architecture. http://www.projectclearwater.org/technical/clearwater-architecture/.

[100] Microsoft Azure Blog. The Windows Azure Malfunction This Weekend. http://azure.microsoft.com/blog/2009/03/17/the-windows-azure-malfunction-this-weekend/.

[101] Microsoft Azure Team. New Windows Server containers and Azure support for Docker.

[102] Microsoft Corporation. Hyper-V.

[103] Microsoft Corporation. Trustworthy computing homepage.

[104] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1).

[105] MoinMoin. The MoinMoin Wiki Engine. http://moinmo.in/.

[106] A. Mukherjee and D.P. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE TSE*, 23(6), 1997.

[107] NASA. NASA Software Safety Guidebook. *NASA-GB-8719.13*, 2004.

[108] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira. On Fault Representativeness of Software Fault Injection. *IEEE Transactions on Software Engineering*, 39(1), 2013.

[109] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):44, 2016.

[110] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3), February 2016.

[111] NFV ISG. Network Functions Virtualisation - An Introduction, Benefits, Enablers, Challenges & Call for Action. Technical report, ETSI, 2012.

[112] NFV ISG. Network Functions Virtualisation (NFV) - Network Operator Perspectives on Industry Progress. Technical report, 2013.

[113] NFV ISG. Network Functions Virtualisation (NFV) - Virtual Network Functions Architecture. Technical report, ETSI, 2013.

[114] NFV ISG. Network Function Virtualisation Infrastructure Architecture - Overview. Technical report, 2014.

[115] NFV ISG. Network Function Virtualisation (NFV) - Resiliency Requirements. Technical report, ETSI, 2014.

[116] NFV ISG. Network Functions Virtualisation (NFV) - Management and Orchestration. Technical report, ETSI, 2014.

[117] NFV ISG. GS NFV-REL 001 - V1.1.1 - Network Functions Virtualisation (NFV); Resiliency Requirements. 2015.

[118] NFV ISG. Network Functions Virtualisation (NFV); Assurance; Report on Active Monitoring and Failure Detection . Technical report, 2016.

[119] NFV ISG. Report on the application of Different Virtualization Technologies. Technical report, 2016.

[120] Openstack. Openstack.

[121] OpenStack. OpenStack Architecture. http://docs.openstack.org/training-guides/content/module001-ch004-openstack-architecture.html.

[122] OpenVZ. OpenVZ Main Page.

[123] D. Oppenheimer, A. Ganapathi, and D.A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *USENIX Symp. on Internet Technologies and Systems*, 2003.

[124] Larry L. Peterson and Bruce S. Davie. *Computer Networks, Fifth Edition: A Systems Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[125] Cuong Pham, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. CloudVal: A framework for validation of virtualization environment in cloud infrastructure. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 189–196. IEEE, June 2011.

[126] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proc. FTCS*, 1992.

[127] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for Fault Tolerance Coverage Evaluation. *IEEE Trans. on Computers*, 44(2), 1995.

[128] Christofer Price and Sandra Rivera. Opnfv: An open platform to accelerate nfv. *White Paper. A Linux Foundation Collaborative Project*, 2012.

[129] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, pages 241–254, 2004.

[130] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum). TL 9000 Quality Management System Measurements Handbook 4.5. Technical report, 2010.

[131] RedHat. Virt-manager. http://virt-manager.et.redhat.com/.

[132] M. Rodríguez, F. Salles, J.C. Fabre, and J. Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. *Proc. EDCC*.

[133] Csaba Rotter, Lóránt Farkas, Gábor Nyíri, Gergely Csatári, László Jánosi, and Róbert Springer. Using linux containers in telecom applications. *Proc. ICIN*, 2016.

[134] SDNCentral LLC. *2016 Mega NFV Report Part I: MANO and NFVI*. https://www.sdxcentral.com/reports/nfv-mano-nfvi-2016-download/, 2016.

[135] Stephen Shankland. Google App Engine suffers outages. http://www.cnet.com/news/google-app-engine-suffers-outages/.

[136] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, May 2010.

[137] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. CCA*, 2008.

[138] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3), 2007.

[139] SPEC. *SPECweb99 v1.02*. http://www.spec.org/web99/, 2000.

[140] SQLAlchemy. SQLAlchemy Homepage. http://www.sqlalchemy.org/.

[141] Standard Performance Evaluation Corporation. SPEC Homepage. https://www.spec.org/.

[142] David T Stott, Greg Ries, Mei-Chen Hsueh, and Ravishankar K Iyer. Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. *Computers, IEEE Transactions on*, 47:108–119, 1998.

[143] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, 1991.

[144] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, volume 21, 1991.

[145] Ali Sunyaev and Stephan Schneider. Cloud services certification. *Communications of the ACM*, 56(2):33–36, 2013.

[146] Nisha Talagala and David Patterson. An analysis of error behaviour in a large storage system. Technical report, 1999.

[147] Technavio. *Global Network Function Virtualization Market 2016-2020.* , 2016.

[148] TPCC. *TPC Benchmark C.* http://www.tpc.org/tpcc/, 2010.

[149] Transaction Processing Performance Council. TPC Homepage. http://www.tpc.org/.

[150] T.K. Tsai and R.K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proc. Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, 1995.

[151] Ariel Tseitlin. The antifragile organization. *Commun. ACM*, 56(8):40–44, August 2013.

[152] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5), 2005.

[153] M. Vieira and H. Madeira. A dependability benchmark for OLTP application environments. In *Proc. Intl. Conf. on Very Large Data Bases*, 2003.

[154] Inc. VMware. VMware vSphere 6 Fault Tolerance - Architecture and Performance. Technical report, 2016.

[155] VMware Inc. VMware ESXi Overview.

[156] VMware, Inc. VMware website. http://www.vmware.com/.

[157] VMware Inc. Delivering high availability in carrier grade nfv infrastructures. *White Paper. VMware vCloud NFV*, 2010.

[158] VMware Inc. vSphere Virtual Machine Administration, 2016. https://www.vmware.com/support/pubs/.

[159] J.M. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly "Good" Software Can Behave. *IEEE Software*, 14(4), 1997.

[160] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[161] Alan Warren. What happened to google docs on wednesday, September 2011.

[162] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. Grinder: on reusability of fault injection tools. In *Proceedings of the 10th International Workshop on Automation of Software Test*, pages 75–79. IEEE Press, 2015.