UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS
IN
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

AUTONOMIC OVERLOAD MANAGEMENT FOR LARGE-SCALE VIRTUALIZED NETWORK FUNCTIONS

STEFANO ROSIELLO

TUTOR: PROF. DOMENICO COTRONEO

COORDINATOR: PROF. DANIELE RICCIO

XXXI CICLO

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

UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

DOCTORAL THESIS

Autonomic Overload Management For Large-Scale Virtualized Network Functions

Author:
Stefano ROSIELLO

Supervisor:
Prof. Domenico COTRONEO

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Information Technology and Electrical Engineering

Scuola Politecnica e delle Scienze di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

.

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Abstract

Doctor of Philosophy

Autonomic Overload Management for Large-Scale Virtualized Network Functions

by Stefano ROSIELLO

The explosion of data traffic in telecommunication networks has been impressive in the last few years. To keep up with the high demand and staying profitable, Telcos are embracing the Network Function Virtualization (NFV) paradigm by shifting from hardware network appliances to software virtual network functions, which are expected to support extremely large scale architectures, providing both high performance and high reliability.

The main objective of this dissertation is to provide frameworks and techniques to enable proper overload detection and mitigation for the emerging virtualized software-based network services. The thesis contribution is threefold. First, it proposes a novel approach to quickly detect performance anomalies in complex and large-scale VNF services. Second, it presents *NFV-Throttle*, an autonomic overload control framework to protect NFV services from overload within a short period of time, allowing to preserve the QoS of traffic flows admitted by network services in response to both traffic spikes (up to 10x the available capacity) and capacity reduction due to infrastructure problems (such as CPU contention). Third, it proposes *DRACO*, to manage overload problems arising in novel large-scale multi-tier applications, such as complex stateful network functions in which the state is spread across modern key-value stores to achieve both scalability and performance. DRACO performs a fine-grained admission control, by tuning the amount and type of traffic according to datastore node dependencies among the tiers (which are dynamically discovered at run-time), and to the current capacity of individual nodes, in order to mitigate overloads and preventing *hot-spots*.

This thesis presents the implementation details and an extensive experimental evaluation for all the above overload management solutions, by means of a virtualized IP Multimedia Subsystem (IMS), which provides modern multimedia services for Telco operators, such as Videoconferencing and VoLTE, and which is one of the top use-cases of the NFV technology.

Contents

Abstract **i**

1 Introduction **3**

1.1 The need of autonomic overload management 3

1.2 Overload management: Threats and challenges 5

1.3 Thesis contributions 9

2 Related Work **13**

2.1 Overload concepts in NFV 13

2.2 Detection of performance anomalies 24

2.3 Cloud elasticity and autonomic capacity scaling 27

2.4 Physical resource contention management 28

2.5 Admission control and traffic throttling strategies 30

2.6 Unbalanced load control in stateful architectures 31

3 On-line detection of performance bottlenecks **33**

3.1 Fault correlation approach 34

3.2 The IMS Case Study 39

3.3 Experimental evaluation 44

3.3.1 Sudden workload surges 48

3.3.2 Component failure 53

3.3.3 Anomaly-free, long-running workload 56

4 Managing the overload of network functions in the Cloud 59
4.1 The problem of overload control in NFV 60
4.2 The proposed overload control solution 62
4.2.1 VNF-level design 65
4.2.2 Host-level design 70
4.2.3 Network-level design 75
4.3 Experimental evaluation on an NFV IMS 78
4.3.1 Testbed and technical implementation 78
4.3.2 Experimental plan 80
4.3.3 Experimental results 83
4.4 Managing the threats of the physical CPU contention at guest-level 92
4.4.1 Overview of CPU overloads and CPU utilization metrics . 92
4.4.2 Mitigation strategy 96
4.4.3 Experimental evaluation 102
4.4.4 Basic feedback control-based overload control 104
4.4.5 Enhanced feedback control-based overload control . . . 107
4.4.6 Performance evaluation under different contention patterns 110

5 Managing the overload of stateful multi-tier network functions 115
5.1 The problem of unbalanced overloads in multi-tier systems . . . 116
5.2 The proposed solution 122
5.2.1 The Distributed Memory component 124
5.2.2 The Distributed Capacity Monitoring 127
5.2.3 The Distributed Admission Control 129
5.3 The Distributed Fileserver Case Study 134
5.3.1 Integration of the overload control solution 137
5.3.2 Experimental evaluation 138
5.4 The IP Multimedia Subsystem Case Study 144
5.4.1 Integration of the overload control solution 146
5.4.2 Experimental evaluation 148
5.5 Overhead and scalability of the overload control solution 156
5.5.1 Sizing the Distributed Memory 158
5.5.2 Example: scaling the solution up to 10K nodes 159
5.5.3 Further optimizations 162

6 Conclusion 165

Bibliography **171**

List of Acronyms

The following acronyms are used throughout this text.

CAPS	Call Attempts per Second
COTS	Commercial Off-the-Shelf components
DNS	Domain Name System
ETSI	European Telecommunications Standards Institute
HSS	Home Subscriber Server
IMS	IP Multimedia Subsystem
IoT	Internet of Things
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
OS	Operating System
P-CSCF	Proxy Call Session Control Function
QoS	Quality of Service
RAPS	Register Attempts per Second
S-CSCF	Serving Call Session Control Function

SLA	Service Level Agreement
VIM	Virtualization Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function

List of Tables

2.1 Service Availability classification levels 20
2.2 Examples of Grades of Service under Different Network Conditions 21

3.1 Clearwater VMs deployment configuration. 43
3.2 Factors and levels for studying the impact of workload surges. . 49
3.3 Detection outcomes and latency under workload surges. 52
3.4 Factors and levels for studying the impact of failure events. . . . 54
3.5 Results for detection based on running correlation for overload conditions due to failures. 56

4.1 Workloads used to evaluate the overload control solution. 83

5.1 Configuration of the experimental Fileserver testbed 140
5.2 Configuration of the experimental IMS testbed 149

List of Figures

2.1 NFV architecture. 14
2.2 IMS S-CSCF transparent failover 16
2.3 Distributed resiliency 17

3.1 A pipeline of network functions 34
3.2 Running correlation between two VNFs 35
3.3 Coefficient of variation filter 40
3.4 Architecture of the Clearwater IMS. 41
3.5 Experimental testbed. 42
3.6 Example of negative running correlation between P-CSCF and S-CSCF CPU utilization. 47
3.7 VNF graph representing the chain of services' utilization. 48
3.8 Registration attempts per minute and registrations completed per minute 50
3.9 Registration attempts and registrations completed per minute, under component failures (due to faults injected at minute 20). 53
3.10 Variable workload below the engineered capacity. 57
3.11 Variable workload that saturates the engineered capacity. 57

4.1 Network throughput under overload conditions. 61
4.2 Overview of the overload control solution. 63
4.3 Architecture of VNF-level detection and mitigation. 66
4.4 Architecture of host-level detection and mitigation. 70

4.5 Architecture of network-level detection and mitigation. 75
4.6 Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Node Level. 84
4.7 Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Network Level. 86
4.8 Mitigation performance at different operational levels (i.e., node, host and network level) 89
4.9 Overload control results at Host level for pCPU contention. . . . 90
4.10 CPU Consumption of the UDP mitigation proxy 91
4.11 CPU Consumption of the TCP mitigation proxy 91
4.12 CPU utilization metrics under three scenarios. 93
4.13 Chain of events caused by physical CPU contention. 98
4.14 CPU utilization metrics under physical contention, with virtual CPU placeholder. 100
4.15 Performance of the IMS registrations during a 2.5x traffic spike (450-900s), using the basic feedback control loop. 105
4.16 Performance of the IMS registrations during CPU contention (300-600s), using the basic feedback control loop. 106
4.17 Virtual CPU utilization during CPU contention (300-600s), using the basic feedback control loop. 106
4.18 Performance of the IMS registrations during a 2.5x workload spike (450-900s), using the enhanced feedback control enabled. 108
4.19 Performance of the IMS registrations during CPU contention (300-600s), using the enhanced feedback control loop. 109
4.20 Virtual CPU utilization during CPU contention (300-600s), using the enhanced feedback control enabled. 109
4.21 IMS Registration (4.21a) and IMS Call-setup (4.21b) throughput during CPU contention, with the basic and the enhanced feedback control. 109
4.22 Cumulative distribution of registration latency, with the basic (red line) and the enhanced (blue line) feedback control. 110
4.23 IMS Throughput (4.23b) and IMS Latency (4.23a) under different CPU contention patterns, with the basic and the enhanced feedback control. 112

5.1 The typical multi-cluster architecture 117

5.2 Distribution of the request rate and the CPU utilization across the data-tier nodes 118
5.3 The overload scenario caused by hot-spot resources 120
5.4 The overload scenario caused by unequal node configurations. . 121
5.5 The overload scenario caused by a transient reduced capacity . 122
5.6 Overview of *DRACO* architecture (red blocks). 125
5.7 The Capacity Monitoring block 127
5.8 The Admission Control process 130
5.9 The location function maps application requests to their location in the storage tier 132
5.10 The architecture of the distributed fileserver casestudy 136
5.11 Resource Location Discovery logic for the distributed fileserver case study 139
5.12 Phases of the evaluation experiments. 141
5.13 Fileserver performance at the engineered capacity (1x) 142
5.14 Fileserver performance at 10x engineered capacity 144
5.15 Summary of the results on the distributed fileserver, with and without the overload control solution 145
5.16 Clearwater IMS Components 146
5.17 Resource Location Discovery logic for the IMS case study 147
5.18 Phases of the evaluation experiments. 150
5.19 Timeseries IMS at the engeenered capacity 153
5.20 Timeseries IMS at 10x the engeenered capacity 154
5.21 Summary of the results with and without the overload control solution, during unbalanced datastore overload 155
5.22 Overhead of the mitigation agent at each load level for the IMS Case Study 157
5.23 Overhead of the mitigation agent at each hotspot scenario for the Fileserver Case Study 158
5.24 Performance of a Distributed Memory server at increasing level of concurrent connections (up to 50K) 160
5.25 Minimum number of Distributed Memory nodes required by the solution, for different scale of application servers (N_A), by varying the average number of storage operations per each user request ($\overline{R}$) 162

Chapter 1

Introduction

1.1 The need of autonomic overload management

The explosion of data traffic in telecommunication networks has been impressive in the last few years. The networks of today connect computers, phones, cars, TV and IoT devices and provide us billion of different services, such as VoIP and instant messaging, gaming and VR, maps, IPTV and video-streaming up to Ultra HD definition. Moreover, pervasive services provided by the giants of the Internet as Google, Apple, Facebook, Amazon, and Netflix are increasing the competition among Telecom operators: customers are constantly pushing for more innovative services and expect them to be provided with a high quality of experience, in their offices, in their homes as well as in mobility through their smartphones and other smart devices.

As result, if in the past telecommunication networks where challenged by exceptional events like a new year or a natural catastrophe (such as an earthquake), nowadays mass events are more frequents: we can think of a viral post on the social networks, the release of a new version of a popular app or an update of the Operating System, a new episode of a TV series, the live streaming of a sport match. The above are only a few examples of challenges for today networks and they cannot be considered exceptional events anymore.

In this context, traditional network architectures, which are complex and hard to scale and to manage, become a real bottleneck for the innovation due to higher maintenance costs and unacceptable higher roll-out times for new services. To keep up with the demand and staying profitable, Telcos are embracing the Network Function Virtualization (NFV) paradigm by shifting from hardware network appliances to virtual network functions, implemented in software. NFV aims to leverage standard IT virtualization technology to consolidate network functions in industry-standard high volume servers, switches, and storage; and to take advantage of orchestration and monitoring solutions used for cloud computing [1, 2].

Being a cloud-based solution, NFV inevitably inherits the threats coming from this domains. A major cause of cloud service failures is represented by overload conditions [3] which occur when the incoming traffic exceeds the available capacity (e.g., by tens, or even hundreds of times). However, overloads are not only due to *traffic spikes* (e.g., due to mass events): an important class of problems in this area is represented by faults that restrict the capacity causing overload, such as faults that can occur in commodity hardware and software components [4, 5, 6, 7], physical resource contention inside the cloud infrastructure [8, 9] and, even more frequently, misconfigurations due to human intervention.

Despite the above threats, the NFV solutions are expected to support extremely large scale architectures, providing high performance and high dependability. Indeed, telecom regulation imposes carrier-grade requirements in term of high packet processing and availability (99.999% or even higher). For this reason, the main consortia behind NFV, including the ETSI and OP-NFV, pointed out the need for solutions capable of:

- optimizing the performance at very large scale without human intervention in response to both service configuration and workload variations [10];

- detecting the occurrences of network problems and mitigating their symptoms within few seconds [11, 12];

The problem of managing the overload conditions touches both these aspects: First, the overload management is responsible to dynamically optimize the resource usage (such as Compute, Memory, Network), in order to prevent both the exhaustion and the under-utilization of physical infrastructure resources, in response to workload changes. Second, it is responsible to guarantee an acceptable QoS by masking or mitigating the effects of faults affecting the service capacity. Moreover, an overload management solution needs to reconfigure itself without human intervention in response to changes in both service workload and scale. Therefore, an effective overload management framework for NFV should be an autonomic solution, in order to react timely to bottlenecks undermining the performance and the availability of the network services.

This dissertation faces the problem of autonomic overload management for virtual network functions, with a case study of a virtualized IP Multimedia Subsystem (IMS), which is, today, according to ETSI [13], one of the network services that will benefit the most from the NFV paradigm.

1.2 Overload management: Threats and challenges

An overload condition occurs when a system has insufficient resources to serve the incoming requests. This condition happens when the current workload hits a bottleneck in one of its components, which limits the capacity of the whole system. A performance bottleneck can arise as a consequence of causes both external and internal to the system. The external ones are due to workload changes both in *intensity*, such as traffic surges causing an incoming traffic exceeding the system capacity, and *type*, such as changes in users behavior causing a bottleneck shift to a lower capacity component of the system. Internal causes are due to factors reducing the available capacity, such as hardware or software failures or misconfigurations, background and maintenance tasks, and contention/interference with other services co-located on the same infrastructures.

When one of the above conditions occurs, the "useful throughput" of the system (i.e., the rate of successfully processed traffic) can significantly degrade; high-priority requests may experience failures; user sessions that were already admitted in the system may be disrupted, causing avalanche restarts and cascade failures due to retries and traffic handover; and handling too much traffic at the same time increases the likelihood of software failures such as failed resource allocations, timeouts, and race conditions. Thus, to achieve an effective overload management in NFV, there are several challenging issues that need to be addressed, both for *overload detection* and for *overload mitigation.*

By looking at the existing literature, the classical approach to detect performance bottlenecks in cloud infrastructures is based on anomaly detection [14]. However, these techniques suffer from limited flexibility, as they require to train classification algorithms with data obtained from extensive test campaigns or with historical data [15, 16, 17]. Although there are few recent studies that adopted these approaches in the context of NFV systems [18, 19, 20], the need for data training could be unattainable for the following reasons. First, **since new service function chains have to be delivered in a short time, it is very difficult to perform test campaigns to get training data**. Second, historical data cannot be used because each service has different characteristics, thus it is very difficult to tailor previous datasets to new contexts. Furthermore, other studies on anomaly detection used threshold-based classifiers, which are easier to deploy. Even in this case, such approaches still need to be calibrated for the specific service, which is very difficult to achieve.

In principle, virtual network functions could take advantage of cloud elasticity by scaling-out network services with on-demand resource allocation to face overload conditions. Unfortunately, cloud elasticity alone is not sufficient to meet the strict high-availability requirements of "*carrier-grade*" telecom services, which often can only afford few tens of seconds of outage per month [21, 22]. As a matter of fact, scaling-out can require up to several minutes to allocate new VM replica [23, 24]; moreover, in the case of extreme overload conditions, an individual cloud datacenter may lack resources for scaling, thus requiring coordinated actions across several datacenters [25, 26]. For these reasons, NFV **requires additional solutions for mitigating overloads in the *short-term*** (i.e., within few tens of seconds), by rejecting or dropping the traffic in excess with respect to the capacity of the network.

Overload management must also take into account the **limited observability and the limited controllability imposed by the "as-a-service" model of cloud computing**, for both Virtual Network Function (VNF) and NFV Infrastructure (NFVI) providers. On the one hand, providers of *VNFaaS* must face the lack of control of the underlying public cloud infrastructures, limiting the opportunities to introduce overload control solutions at the infrastructure level. On the other hand, NFVI providers have little visibility and control on VNF software, since it will be distributed and deployed as black-box VM images on their *NFVIaaS* [27]. In this case, overload control should not rely on the cooperation of VNF software.

Another important, and often underestimated, cause of overload conditions is the *resource contention* inside the cloud infrastructure, whose effect is to decrease the available capacity for serving the incoming traffic. The resource contention has severe side effects on time-critical applications that run at the guest level (i.e., inside VMs, such a VNFs). As the first effect, the application receives less virtual resources (e.g., less virtual CPU time) than the resource quota agreed with the cloud infrastructure provider, which leads to performance degradation and service failures. The second (and more subtle) effect is that guest OS resource utilization metrics (in particular the virtual CPU utilization) can mislead load control mechanisms inside the guest, such as real-time rate adaptation [28], graceful performance degradation through brown-out [29] and traffic shaping [30, 31]. Thus, overload management solutions **should be aware of the contention phenomena and provide mitigation both at infrastructure and at service level**.

Moreover, complex network functions are typically implemented as multi-tier systems. Traditionally, this division enforces low coupling between tiers, high cohesion within them, and agnosticism of consumers [32]. More recently, these principles of transparency and decoupling have been challenged by the extremely large scale reached computing systems. Highly scalable network function implementations, such as the vIMS, are organized as a set of stateless microservices keeping the state of the application nodes, in one or more separate storage tiers. This approach is enabled by new highly-distributed NoSQL datastores, such as Cassandra and Memcached [33], which can balance the storage load across thousand of nodes using techniques such as consistent hashing [34] and key-range topologies [35].

The above picture hide new threats for the overload management since bottlenecks can shift from one tier to another as workload pattern changes. Moreover, in large clusters, bottlenecks can arise because of a small group of nodes, forming an unbalanced load condition within the same tier called "hot-spot" [36, 37, 38]. When this occurs, even if there is still available capacity in each tier, the system is exposed to the risk of overload. Therefore, overload control solution in large multi-tier systems must face two additional challenges:

1. **Traffic can only be throttled by the application tier**. Typically, an ap-

plication session involves a series of requests to several storage nodes. In order to assure data consistency, the storage tier should not drop any traffic in the middle of an application session; or, application transactions should be rolled-back, resulting in a waste of resources and in additional application complexity. Therefore, the traffic in excess should only be filtered in the outer tier even if the bottleneck is in an inner tier.

2. **Traffic throttling must account for data location dependencies**. Since key-value stores distribute and retrieve resources using consistent hashing, the storage tier can experience a *unbalanced* overload condition that affects a subset of storage nodes, for example in the case of "hot-spot" resources that are requested at an unexpectedly-high rate. Moreover, it is possible that only specific nodes in the persistence tier are affected by unbalanced overload conditions, because of resource exhaustion or competition against other services due to a software bug, a failed update, wrong configuration, or over-commitment.

Unfortunately, as discussed in the Chapter 2, the existing solutions, such as limiting the incoming traffic according to the available capacity of a tier [39, 40, 41] or adding new resources by scaling up the tier [42, 43], do not consider resource dependencies between the tiers, leading to severe inefficiencies in the case of unbalanced overloads, since the capacity of even large multi-tier system may be limited by the capacity of few nodes.

1.3 Thesis contributions

The main objective of this dissertation is to provide frameworks and techniques to enable proper overload detection and mitigation for the emerging virtualized software-based network services.

Chapter 3 presents a novel approach to identify *performance anomalies* in

NFV services composed as a chain of network functions. The key feature is that the approach neither requires to train a model nor to calibrate a threshold to identify performance anomalies inside the VNF chain. Instead, the proposed approach takes advantage from the fact that the VNF chain can be seen as a multistage pipeline, where the output of a network function is the input of the next one. Therefore, the resource utilization metrics of VNFs in a service chain have a strong dependency (e.g., the outgoing network traffic from the first stage is related to the CPU load on the second stage of the chain). Thus, the approach collects metrics from connected VNF stages, as they are naturally correlated. Then, it analyzes their co-variation over time to infer potential performance anomaly at each stage of the chain. The approach can also be adopted in large-scale NFV systems with the presence of load-balancing and replication.

Chapter 4 presents a novel overload control framework for NFV (*NFV-Throttle*) to protect NFV services from overloads within a short period of time, by tuning the incoming traffic towards VNFs in order to make the best use of the available capacity, and to preserve the QoS of traffic flows admitted by the network services. The framework consists of a set of modular *agents*, which detect an overload condition (either of individual VNFs and hosts, or of the NFV network as a whole), and mitigate it by dropping or rejecting the traffic in excess, and by tuning resource allocation to VNFs according to their priority, in order to relieve physical resource contention. The agents can be installed either at VNF-level (for *VNFaaS* providers) or at NFVI-level (for *NFVI-aaS* providers), without requiring changes to VNF software, and fitting into the *as-a-service* model. The key idea of the proposed solution is to **protect the application at the guest-level** (i.e., by running inside a VM), and to be **complementary to recovery mechanisms at the infrastructure-level**. Such an approach is especially relevant in the case of **infrastructures-as-a-service** (IaaS), where a time-critical application (e.g., a VNF) has little visibility and control on the underlying physical resources (e.g., on scheduling priorities at the physical CPU level). To the best of our knowledge, no previous work has addressed the problem of physical contention from this perspective. Moreover, the (*NFV-Throttle*) framework studies the overloads due to physical CPU contention, by showing that it can cause unpredictable side effects on the QoS of time-critical applications. Chapter 4 presents a generalized overload control solution, based on traffic-throttling, which includes overload conditions caused by physical CPU contention. The contention-aware *feedback-loop based throttling solution*, checks at run-time the resources that are currently available for the VM, and only accepts a portion of the traffic that can be served with an adequate quality of service [31, 44, 45].

In Chapter 5, the thesis proposes a solution to the overload control problems arising in virtual network function implemented as large-scale multi-tier applications. It presents *DRACO* (*Distributed Resource-aware Admission COntrol*), a novel autonomic solution that addresses overload problems arising in any tier of the system. It performs a fine-grained admission control, by tuning the amount and type of traffic according to resource dependencies among the tiers (which are dynamically discovered at run-time), and to the current capacity of individual nodes, in order to mitigate overloads while achieving a high resource utilization. Moreover, the solution acts solely at the application service interface, in order not to impact on data consistency and to preserve the highly distributed and scalable architecture of multi-tier systems.

This thesis presents an extensive experimental evaluation for all the above overload management solutions, including the anomaly detection framework, *NFV-Throttle* and *DRACO*, by means of the Project Clearwater [46], an open-source implementation of a Virtualized IMS, also commercially supported by Metaswitch Networks Inc. This thesis also benefits from the fruitful collaboration with industry: the experimental evaluations are performed at different scale (up to hundreds nodes) and with different service configurations in order to reproduce overload conditions which are representative of problems occurring in real-world NFV infrastructures, or to emphasize pathological conditions of the NFV software stack.

The work includes material from the following research papers, already accepted or published in peer-reviewed conferences and international journals:

- D. Cotroneo, R. Natella, S. Rosiello. *"A fault correlation approach to detect performance anomalies in Virtual Network Function chains"*, Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on. IEEE, 2017.
- D. Cotroneo, R. Natella, S. Rosiello. *"NFV-Throttle: An Overload Control Framework for Network Function Virtualization"*, IEEE Transactions on Network and Service Management 14.4 (2017): 949-963.

Chapter 2

Related Work

2.1 Overload concepts in NFV

In traditional network functions (NF), the levels of performance and reliability are well-known and understood. In NFV, traditional (hardware-based) network functions will be superseded by network functions implemented in software and leveraging virtualization technologies. NFV promises to reduce costs, improve manageability, reduce time-to-market, and provide more advanced services [47].

Figure 2.1 shows the architecture of an NFV system. It is characterized by three main components.

- **Virtualized Network Functions (VNFs)** are the software-implemented network functions used to process the network traffic (according to some network protocol and network topology). They use both virtual and physical resources.
- **NFV Infrastructure (NFVI)** abstracts and manages access to physical resources. It includes the hardware resources, a virtualization layer to cre-

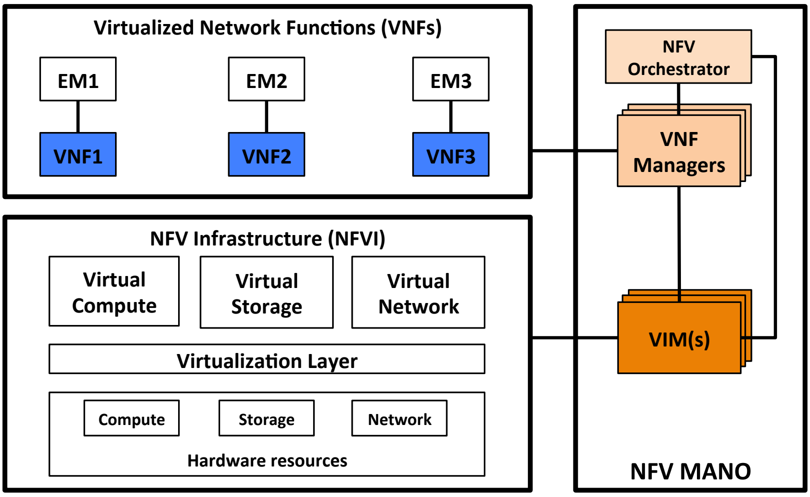


Figure 2.1. NFV architecture.

ate virtual resources on the available hardware, and the virtual resources themselves.

- **NFV Management and Orchestration (MANO)** acts as coordinator/orchestrator of the overall NFV system. It includes three types of sub-components. An *orchestrator*, which allocates and releases resources of the NFVI to the VNFs, by using the VIM, and manages the lifecycle of network services (NS) (creation, scaling, configuration, upgrading, termination). *VNF managers* are used to manage the lifecycle of VNFs. Each VNF is linked to a VNF manager. *Virtualised Infrastructure Managers (VIMs)* are controlled by the NFV Orchestrator and VNF Managers to manage physical and virtual resources in the NFVI. A VIM is not aware of VNFs executed in the VM.

A possible issue of the NFV approach is that virtualization adds more complexity and new risks, which will threaten the performance and reliability of the whole network infrastructure. Indeed, Telecom services are expected to be always available and, as soon as a failure or an outage occur, they must be recovered within a short period of time (e.g., milliseconds), using automatic recovery means.

If we think about legacy networks, overload can occur both in a physical node (i.e., network function) and in a physical link. Commonly, network functions (e.g., firewalls, load balancers, routers, switches, etc.) are considered as overloaded when their limited capacity, seen as the set of hardware (e.g., cpu, memory) and software resources, is exceeded due to huge number of requests. Furthermore, overload may occur also at physical links in the network when the incoming traffic exceeds the link bandwidth. [48]

In general, we have to consider *network congestion* that occurs when an high number of requests are submitted to the network infrastructure. Such a situation overcomes the total **capacity** of the infrastructure, and it may lead to an overall degradation of the network as a whole; for instance, the throughput may decrease, and the latency and jitter increase. Thus, shifting from hardware-based network function to software-based network function is a big challenge.

In NFV, the problem of overload is discussed by ETSI within the more general problem of *resiliency*[1].

ETSI GS NFV-REL (2015) [50] 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 the document, ETSI addresses the NFV resiliency problem by imposing the following design criteria:

- Service continuity and failure containment;
- Automate recovery from failures;

[1] In general, resiliency is the capability of a system to adapt itself properly when facing faults or changes in the environment [49]

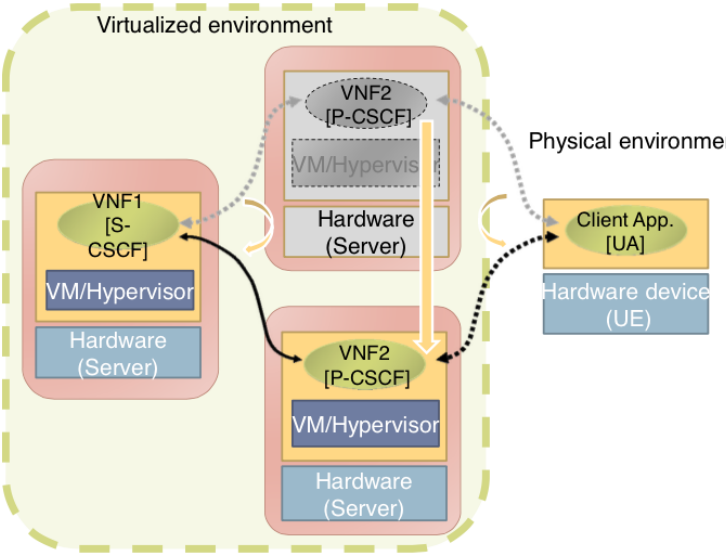


Figure 2.2. IMS S-CSCF transparent failover

- Prevent single point of failure in the underlying architecture;
- Multi-vendor environment;
- Hybrid Infrastructure.

The document also introduces the *service continuity* as the capability of assuring quality of service goals when anomaly conditions, such as the overload ones, occur. The NFV will have to provide a continuous service even if a component fails, or incoming service requests are much more than the norm. For example, in the IP Multimedia Subsystem (IMS) use case identified by the ETSI NFVI Industry Specification Group (ISG) [51], a critical scenario that can lead to an overload condition of the IMS is when a crucial component (such as the S-CSCF registrar server) fails; this situation may cause a potential (re)connection or signalling storm that overloads the whole infrastructure.

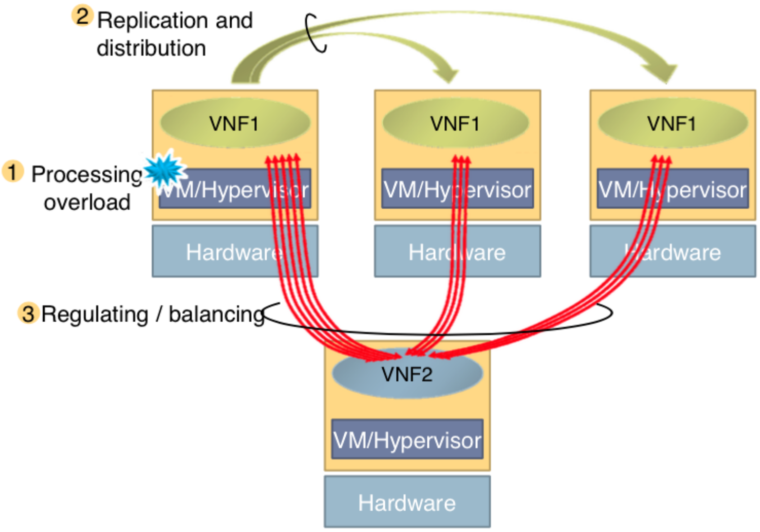


Figure 2.3. Distributed resiliency

Figure 2.2 shows that a Virtualized Network Function (VNF) failure should be recovered in a transparent way. Figure 2.3 shows that it is very critical to balance, in a seamless way, the traffic of requests from overloaded/failed VNF to other VNFs, residing on different hosts.

Another aspect of resiliency is the *service availability level* of NFs. The virtualized NFs must guarantee that the provided quality of service is the same as hardware-based NFs (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 [52]. The ETSI NFV scenarios envision the emergence 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 will require mechanisms for storing and recovering the state of network sessions and connections in a reliable way.

In detail, 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 VNFs' resources. This situation may result from special events (e.g., Chinese New Year festival, TV shows, etc.) or from a cyber-attack (e.g., DDoS attack). In *failure conditions*, a service may be interrupted or it becomes unavailable due to faulty components. In that situation, two key factors play a crucial role: priority of restoration and failure recovery time. *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. *Failure recovery time* is the time needed to recovery from failures, instead, and it depends on the amount of redundant resources available. 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.

According to the NFV resiliency requirements [50], in such situations, the NFV will need to assure the availability of deployed services according to well-defined *service availability levels*, which are agreed with customers and/or imposed by regulations. An availability level specifies the importance of a service and the redundancy that will need to be provided for that service. For instance, Table 2.1, adapted from [50], describes three availability levels, and the services under each availability level: Level 1 includes the services with the most stringent requirements (such as emergency telecommunication) and prescribes the use of the highest level of redundancy (1+1 with instantaneous switchover); Level 3 includes the less critical services (such as general data traffic) and the less stringent requirements (e.g., best effort service with M+1 redundancy).

Given the varying resiliency needs, the NFV is expected to give priority to the

most critical services in the case of overload conditions. The ETSI resiliency requirements document provides an example of this behavior. Table 2.2 provides a list of services (Video call, gaming, financial transaction) and their service availability levels; moreover, the table describes the grades of service of the NFV infrastructure in the presence of different network conditions, including: normal condition, overloaded condition, heavily overloaded condition, and emergency situation.
In normal conditions, a video call service can provide both video and audio facilities. When overload conditions arise, the video call service is downgraded to a voice call service with static image only, since resources become scarce. Furthermore, if the overload conditions worsen, the NFV must address an emergency situation, in which it is necessary to keep available the most critical service (call service) with at least voice call capability. In the example, the video call service is further downgraded to a voice call service only.

Beyond the NFV resiliency requirements document [50], the reader might guess that in NFV we have to consider the same overload definitions as in legacy networks. Currently, there is no precise definition of overload in NFV standard documents. In providing a definition, we have to consider the different layers of the NFV architecture (see [53]). Specifically,

NFV is overloaded when it has to process a high demand of resources that exceed its limited capacity. Such a demand may saturate processing resources within the VNF layer (i.e., the set of VNFs that provide the specific service), virtual resources within the VM/Hypervisor layer (i.e., the virtual environment that hosts VNF, and allows the sharing of the underlying physical resources among the VNFs), resources within the NFV Management and Orchestration layer (i.e., the components that orchestrate the NFV infrastructure and software resources), or physical resources within the Physical layer (e.g., physical compute, storage, and network resources).

Overload management approaches are specialized for NFV as follows:

Table 2.1. Service Availability classification levels

Availability Level	Customer Type	Service/Function Type	Notes
Level 1	• Network Operator Control Traffic • Government/Regulatory Emergency Services	• Intra-carrier engineering traffic • Emergency tele- communication service (emergency response, emergency dispatch) • Critical Network Infra- structure Functions (e.g. VoLTE functions, DNS Servers, etc.)	Sub-levels within Level 1 may be created by the Network Operator depending on Customer demands. E.g.: • 1A - Control • 1B - Real-time • 1C - Data May require 1+1 Redundancy with Instantaneous Switchover
Level 2	• Enterprise and/or large- scale customers (e.g. Corporations, University) • Network Operators (Tier 1/2/3) service traffic	• VPN • Real-time traffic (Voice and video) • Network Infrastructure Functions supporting Level 2 services (e.g. VPN servers, Corporate Web/Mail servers)	Sub-levels within Level 2 may be created by the Network Operator depending on Customer demands. E.g.: • 2A - VPN • 2B - Real-time • 2C - Data May require 1:1 Redundancy with Fast (maybe Instantaneous) Switchover
Level 3	General Consumer Public and ISP Traffic	• Data traffic (including voice and video traffic provided by OTT) • Network Infrastructure Functions supporting Level 3 services	While this is typically considered to be "Best Effort" traffic, it is expected that Network Operators will devote sufficient resources to assure "satisfactory" levels of availability. This level of service may be pre-empted by those with higher levels of Service Availability. May require M+1 Redundancy with Fast Switchover; where M > 1 and the value of M to be determined by further study

Table 2.2. Examples of Grades of Service under Different Network Conditions

Service name [default SA level]		Normal	Overloaded	Heavily Overloaded	Emergency Situation [dedicated SA level]
Video call service	Video [2]	available [2-I]	available [2-I]	Degraded to Image service [2-II]*	Not available (pre-empted) [2-III]*
	Voice [1] (registered a ETS)	available	available	available	available
Gaming [3]		available	Not available (pre-empted)	Not available (pre-empted)	Not available (pre-empted)
Financial Transaction [1]		available	available	available	Not available (pre-empted) [3]
NOTE: * indicates that the Grade of Service is changed/reduced due to changes in the network status.					

- **Prevention** mechanisms, with the aim to prevent the occurrence of congestion situations, which may be due to VNF's resources saturations, and result in the reduction of the work performed per time unit (throughput) and in the slowing down in responding to service requests (response time).

- **Detection** mechanisms, which monitor the NFV to detect the overload condition. Such mechanisms can be hardware- or software-implemented, and monitoring can be local or distributed among VNFs and the overall NFVI. Furthermore, such mechanisms can act at different granularity, that is at service level, or at infrastructure level. The latency of the detection, i.e. the time from the occurrence of an overload situation to its handling, clearly is an important characteristic of the mechanisms. There could be services for which availability requirements are stringent and require to be recovered as fast as possible.

- **Control** mechanisms, with the aim to handle congestion problems. Mitigation means that even though the capacity of the NFV is exceeded, the service QoS can be decreased but not beyond a certain threshold.

> For example, control mechanisms allow dropping some incoming requests or lowering the bit rate of the communication. Once an overload condition is detected, control mechanisms try to limit the performance degradation by executing actions with the goals to keep the network performance (e.g., throughput, latency) within a specific operational range.

Overload management mechanisms have to consider specific **goals**, for example which are the minimum thresholds of latency or throughput to be kept during overload conditions. To achieve such goals several **actions** can be performed, including:

- *Reconfiguration* actions, that is *migrate* (relocating and restoring its state) a specific VNF, in order to keep the service continuity;
- *Scalability* actions, in order to distribute the load. Such mechanisms consist in the provision of additional resources for a specific VNF. These resources could be taken from idle resource within a different VNF or even a different geo-located NFVI;
- *Filtering* actions that allow the management of incoming traffic that may lead to network congestion.

Overload conditions in NFV need to be addressed both at VNF and NFVI level, and both on virtual and physical resources. For example, overload conditions can lead to virtual or physical CPU overload, connection loss, network latency increase, memory leaks, and deadlock. Parameters that indicates overload situations are both related to VNF/VM load, and the hypervisor load that may impact on VNFs.

At first glance, elastic cloud management techniques appear to solve the overload problem, but differing from the IT domain, network traffic in telecom domain is highly dynamic and too difficult to predict. Thus, elastic resource mechanisms are good to prevent overload situations only when traffic increment does not exceed a specific threshold; otherwise, such mechanisms are not effective, since can not handle network VNFs will experience traffic volumes that suddenly increase in a short period of time. However, elastic reconfiguration mechanisms are not suitable, and not meant, for providing very high availability and performance levels (e.g., availability of 99.99% or more, and response times in the order of milliseconds). Such objectives require very quick reconfiguration mechanisms, which should be aware of the priority of VNFs and services. This is the case of a failed VNF that maintains connection state within million of users. When it is replaced as a result of elastic mechanisms with an new VNF, a non-negligible number of VNFs may need to reconnect within the new VNF, but such reconnection process lead to traffic storm, both at VNF level and at Hypervisor level. Another mechanism that can exacerbate the problem of overload is traffic migration: if traffic is migrated to an already-overload VNF, such migration process may lead to create new overload conditions to other VNFs.

Concurrently to this work, besides the ETSI recomendation, the research projects H2020 Next Generation Platform-as-a-Service (NGPaaS) [54] and SONATA [55] aim to fill the gap between the cloud computing models and the network function virtualization framework, towards “telco-grade” quality for virtual network services in the context of the next 5G communication services. While these projects focus more on the architectural problems, proposing solutions to ease the put in operation and the management of large-scale services and new development processes such as Dev-for-Operations [56], this thesis has a focus on service level reliability and service quality management. However, there are many points in common with the work in this thesis. Chapter 4 discusses some architectural limitations of the cloud models in the context of NFV (such as NFVaaS and NFVIaaS), such as the limited observability and controllability of system parts in case of the separation of infrastructure providers, service providers and network function service developers. The NGPaaS is a new service model which enforce the separation between these entities but extends the management and orchestration components with specialized APIs to overcome the observability and controllability problems. However future virtual network function software needs to be designed and developed according to this framework. On the contrary, this thesis proposes a framework (NFV-Throttle), which adds monitoring/controlling agents in specific points of the NFV architecture (i.e., guest-vnf, host-nfvi and vnf-tenant levels) and these agents communicate by using standard host-guest interfaces (such as hypercalls or vmci-sockets). The advantage is that this approach does not require changes both to the cloud infrastructure nor to the virtual network function software. The same principles inspired also the DRACO framework, presented in Chapter 5.

As discussed in this section, the problem of overload in NFV is still open, and it includes research challenges and problems related to reliability, performance and more in general to resiliency.
In the following sections, there are presented studies proposing approaches that may be exploited to address the problem of overload in NFV.

2.2 Detection of performance anomalies

Continuous, online monitoring and analysis is a key component for managing cloud infrastructures. The analysis of performance metrics and resource utilization enables a better understanding of application and system behaviour, helps to tune configurations to meet application SLA requirements, and provides insights for troubleshooting.

The most common cloud monitoring and dashboards systems, such as Amazon CloudWatch [57] and Google StackDriver [58] monitor the system on per-VM basis and allow to setup and customize simple detection rules (e.g., thresholds on monitored metrics) and trigger maintenance task (e.g., scaling, rebooting). More advanced commercial products, such as Datadog [59], also implement simple data mining features, using seasonal auto regression, trend detection, online adaptive learning, and statistical distribution models. However, since the products focus on symptoms on individual VM instances, they are prone to false alarms: for instance, without any knowledge about the specific applications, they cannot discern if a drop in the load of a VM is caused by a sudden workload decrease in the whole system or by an undetected fault in some component. As discussed later, our approach takes into account the nature of NFV applications (based on pipeline processing of high-volumes of packet streams) to detect these scenarios: it analyzes the correlation of metrics from neighbour VMs in the VNF service chain, to distinguish licit workload variations from faulty conditions that affect the quality of service.

In general, anomaly detection systems aim to automate the discovery and classification of problems by analyzing these data, and checking whether the system behaves accordingly to what is expected. In order to characterize such behaviors, classical approaches use machine learning techniques, such as random forests classifiers [60], neural networks [16], automatic rule learning and fuzzy logic [15], unsupervised clustering [61, 62]. Most of the anomaly detection research has applications in intrusion and misuse detection.

More recently, these approaches have been applied in the context of NFV applications. Miyazawa et al. [18] proposed a distributed architecture to perform fault detection using unsupervised data clustering techniques and self-organizing maps. In [20], Sauvanaud et al. suggest a supervised learning approach. They perform fault injection experiments in a NFV testbed to collect labeled monitoring data, from both hypervisors and virtual machines instances. Then, they build a classifier using the random forest algorithm, showing high detection accuracy and low false positive. However, both approaches require retraining the models in case of changes in the hardware or software configuration, workload patterns or other influencing factors. A recent study assessed these problems by proposing the StepWise framework [], which is able to detect significant changes in data distribution (i.e., concept drifts) that are not anomalies but due to changes in the system configuration on the user behaviour. This approach, could be used in context of NFV to prevent false alarms during common scaling-in or scaling-out operations.

Unfortunately, all the previous techniques require training models with data coming from extensive test campaign or historical data. However, in the context of NFV, the needed training data may be unattainable, since service function chains must be delivered in a short time (thus limiting the amount of tests for getting training data) and are tailored for each specific service (thus limiting the usefulness of historical data). For the same reasons, most anomaly detection systems used in practice are threshold-based classifiers, which are ease to deploy and provide an acceptable quality of detection (in terms of accuracy and latency), but they still need to be calibrated for the specific service.

Contemporary to this work, Schmidt et al. [63] proposed an unsupervised anomaly detection framework tailored for NFV services, which is based on on-line monitoring data and dynamic threshold learning. The approach has been experimentally evaluated by means of anomaly injection experiments and show high accuracy and a low false alarm rate.

In the field of classical (i.e., non-virtualized) network management systems, *alarm correlation* [64] between multiple distributed entities is widely used to detect faults and isolate the causes across a big number of network appliances interconnected [65, 66]. In [67], Kliger et al. defined the networking graph as a causality graph on which nodes can be marked as *problems* or *symptoms*, and use event correlation to find causal relations among the events. They demonstrate that this approach is resilient to high rates of symptom loss (i.e., false negatives) and false alarms. Similarly, this thesis show that it is possible to identify causal relationships in the VNF service chaining model between the VNF instances in the network. We apply the correlation analysis to the monitoring data to recognize symptoms of problems in the network.

2.3 Cloud elasticity and autonomic capacity scaling

One opportunity to face overload conditions in cloud computing is scaling up the architecture, either in a proactive or a reactive way. *Autoscale* [42] is an autonomic solution for modern multi-tier architectures that goes in this direction: application nodes are automatically scaled out in response to workload pattern changes. However, this framework focuses on stateless bottlenecks only. Indeed, scaling stateful datastore tiers can require complex data redistribution and the time taken by a new instance for joining an existing cluster increases during an existing overload condition. Recent work proposed new techniques for scaling out datastores. *PAX* [43] is an approach to scale-out a distributed datastore (i.e., *Cassandra*) that accelerates the distribution of hotspot data partitions on newly added nodes by performing workload profiling to detect resources that are accessed more frequently. However, these capacity optimization solutions designed for cloud computing are not enough to protect virtual network functions for two reasons: the first is that scaling out requires time, during which the system is exposed to a degraded quality of service, cascading component failures due to resource exhaustion. In this timeframe our solution can cooperate with scaling solutions to preserve the quality of service of already established sessions up to the system capacity, while more capacity is added in background to pick up workload variations; the second reason is that overload is not only caused by external workload surges but can be a consequence of software bugs, misconfigurations, poor load balancing or transient management tasks running within the same infrastructure.

In these cases, scaling up can be ineffective or even exacerbate the overload problems. Moreover, scaling out is not always possible since it is constrained by costs and resource availability.

2.4 Physical resource contention management

CPU contention and, more in general, resource contention are typical problems happening in virtualization infrastructures and suffered by guest VMs. Nikounia et al. [68] characterized the performance degradation due to resource overcommittment in virtualized environments. Their study identified the CPU resource as the one that impacts the most on service performance during contention with noisy neighbors VMs, and found a major case of execution time slowdown in the hypervisor CPU scheduler.

Since the problem is widespread in virtualized environments, there have been many studies on ensuring performance isolation at infrastructure level, in order to avoid side-effects from CPU contention. In general, these solutions either prevent or mitigate contention by enhancing the placement and scheduling of VMs on the physical infrastructure. *Q-Clouds* [69] is a representative solution of this kind, which is a QoS-aware framework aiming to enforce performance isolation by opportunistically provisioning additional resources to alleviate contention. Caglar et al. [70] proposed *HALT*, a performance-interference aware placement strategy based on on-line monitoring and machine learning. To avoid the side effects of the contention for time-sensitive services, HALT proposes a VM migration plan to a different host, based on the learned workload behavior. More recently, in the context of NFV, Kulkarni et al. [71] presented *NFVnice*, a framework to dynamically adjust the scheduling behavior according to the relative priority of the running services and the estimated load. This approach uses *cgroups* to optimize the scheduling behavior and traffic throttling at host level to prevent overloads in the guest.

It is important to note that these solutions require full control of the underlying infrastructure (e.g., they are meant for system administrators and infrastructure management products). However, as discussed in this this thesis, contention issues at infrastructure level cannot be avoided completely, due to faults, unexpected maintenance tasks, and misconfigurations. Thus, services with very high-availability requirements need to include cautionary mechanisms to mitigate such scenarios. Moreover, in the case of *NFV Infrastructures as a Service* (NFVIaaS), the VNFs do not have control on the underlying infrastructure, where the infrastructure provider may adopt an over-commitment policy that increases the risk of physical CPU contention, at the expense of the VNFs.

At guest level, the *steal time* metric is a well-known indicator of physical CPU contention. This indicator is typically exposed by hypervisors to the guest OSes. Ayodele et al. [72] demonstrated the impact of the steal time on cloud applications performance under physical CPU contention. Moreover, other studies focus on quantifying the effect of the steal time on CPU time metrics at process- and thread-level [73, 74], provided by the guest OS. Unfortunately, this metric is often adopted in *unsound* heuristics, such as to trigger VM migration when the steal time is very high for a prolonged period [75]. However, a high steal time is not a sufficient condition for a physical CPU contention. Indeed, a VM which is voluntary suspending to perform I/O activity can be subject to high wait time due to the contention with other VMs (e.g., running a CPU-bound workload). In this case, the guest OS metrics will report a lower CPU utilization and low steal time. VMware, for example, suggests not to trust CPU consumption metrics provided by the guest OS as they can be inaccurate in case of physical CPU contention [76] due to time accounting issues. Additionally, even in case of CPU-bound workloads, the steal time can also be inaccurate in case of hyper-threading enabled at host level [77]. Thus, the percentage of steal time is dependent by the workload running in the guest

VM. Moreover, a steal time quota can be the consequence of CPU quotas and CPU credits imposed by the infrastructure providers [78].

2.5 Admission control and traffic throttling strategies

Admission control and traffic throttling solutions have been frequently used in IT and telecom systems to promptly react to overload conditions. In general, these approaches monitor service performance (e.g., in terms of throughput and latency at the application layer) and resource consumption (e.g., CPU utilization), and throttle the traffic according to a dynamic feedback on the available capacity. For example, *Welsh et al.* [31] proposed an adaptive overload control approach using a *token bucket* and a closed control loop to dynamically tune the traffic according to the service latency. *Kasera et al.* [30] analyzed throttling algorithms in the context of carrier-grade telecom switches: the *Random Early Discard* (RED [79]) throttles traffic according to the request queue size, while the *Occupancy* algorithm ensures a target CPU utilization by throttling the traffic according to the CPU utilization and the rate of accepted calls. A similar algorithm has also been applied in the context of virtual network functions by *NFV-Throttle* [80]. *Hong et al.* [45] present a broad overview of these schemes for overload control for the SIP protocol. Lately, these admission control systems have been applied in the context of traditional three-tier web server applications with strict Service Level Objectives, such as e-commerce platforms. *Liu et al.* [39] propose an admission control technique based on the combination of queuing theory models and feedback control loop to perform adaptive load control in these architectures. In case of degraded quality, such as increased latency, the solution discards a percentage of incoming requests. Unfortunately, a solution of this kind suffers from poor performance in newly highly distributed architectures. Indeed, in case of an unbalanced overload in few nodes of a large architecture, rejecting a percentage of random traffic causes the filtering of many requests that would require not overloaded nodes and does not imply filtering requests directed to overloaded ones. *CoSAC* [41] is a different approach to perform Session Based Admission Control [40] in context of multi-tier web applications. *CoSAC* considers that a different incoming request mix (e.g., due to users behaviors) causes the shift from a bottleneck tier to another during the time, and uses a Bayesian network to correlate the state of the application tiers, to perform an admission decision. However, also this approach suffers from similar problems, since it considers the tiers as a whole, it cannot be applied in modern large scale architecture in which a single node can become a bottleneck for an underutilized tier of thousand nodes.

2.6 Unbalanced load control in stateful architectures

In addition to scaling solutions, other studies focus on the load balancing optimization in the datastore tiers to prevent unbalanced overload conditions. These solutions aim to solve the problem within the datastore tier, by mitigating load unbalance with dynamic replication and data migration strategies. *SPORE* [37] is a solution to hot-spot problems due to a highly skewed workload. *SPORE* modifies the traditional Memcached behavior implementing advanced data replication strategies based on key popularity. *Zhang et al.* [81] propose a solution to load imbalance due to a hot-spot workload and server heterogeneity. They designed a middleware component that modifies the way in which data is accessed and distributed in key-value stores (such as Memcached). This component includes a hot-spot detector which takes account of key request frequencies, and a key redirector module that can either replicate the key on multiple servers (e.g., proportionally to its request frequency, similarly to SPORE), or select some keys to be forwarded to servers less loaded or more powerful. *NetKV* [82], is an accelerated proxy to inspect key requests and analyze the datastore workload to replicate hot-spot keys on multiple servers, in order to limit the load unbalancing due to the workload skewness. *MBal* [83] is a novel in-memory datastore architecture aiming to resolve load unbalancing problems within the datastore tier itself. This architecture includes a centralized coordinator that monitor the system state and applies data replication and migration strategies among not only the distributed instances but also at thread level among the CPU cores within each node. However, none of the previous solutions act at system level, by preventing an excess of unbalanced traffic from entering into the system. Dynamic replication and migrations can cause an additional unpredictable amount of load among datastore nodes, and they require additional space and time to handle consistency that could be not available in some circumstances.

Chapter 3

On-line detection of performance bottlenecks

Network Function Virtualization is an emerging paradigm to allow the creation, at software level, of complex network services by composing simpler ones. However, this paradigm shift exposes network services to faults and bottlenecks in the complex software virtualization infrastructure they rely on. Thus, NFV services require effective anomaly detection systems to detect the occurrence of network problems. This chapter proposes a novel approach to ease the adoption of anomaly detection in production NFV services, by avoiding the need to train a model or to calibrate a threshold. The approach infers the service health status by collecting metrics from multiple elements in the NFV service chain, and by analyzing their (lack of) correlation over the time. The approach has been validated on an NFV-oriented Interactive Multimedia System, to detect problems affecting the quality of service, such as the overload, component crashes, avalanche restarts and physical resource contention.

3.1 Fault correlation approach

Our approach is based on the idea that a network packet or request follows a chain of VNFs, as shown by the VNF graph in Figure 3.1. Each VNF in the graph can have a different number of replicas, that are scaled according to a preliminary capacity planning or to cloud elasticity. The load is balanced across all the replicas of the VNF.

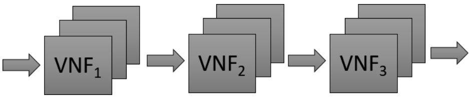


Figure 3.1. A pipeline of network functions

In this architecture, there are metrics from multiple stages that are naturally correlated (e.g., the outgoing network traffic of the first stage and the CPU load of the second stage). If resource utilization (e.g., CPU, memory, ...) increases in a VM hosting a network function, an increase should also occur in VMs hosting the subsequent VNF in the service chain. If this is not the case, a VM is obstructing the network flow, causing a performance anomaly. Thus, we analyze the correlation in the time between metrics from two distinct network functions to infer the service health status. Figure 3.2 shows the vCPU load of two connected network functions (i.e., the output traffic of VNF$^{(A)}$ is processed by VNF$^{(B)}$). When VNF$^{(A)}$ uses all the available CPU time (e.g., due to an overload condition or a software fault), its throughput start decreasing. As a consequence, the VNF$^{(B)}$ receives less traffic to process and the CPU load on this VNF decreases. This condition can be detected noting that there is a window of time in which the CPU load on VNF$^{(A)}$ increases and the CPU load on VNF$^{(B)}$ decreases. In correspondence of that window, the two time series become negatively correlated. We consider this condition a symptom of a performance anomaly.

The algorithm 1 raises an alarm when an anomaly is detected between a pair of connected VNF stages, namely VNF$^{(A)}$ and VNF$^{(B)}$, with VNF$^{(A)}$ preceding VNF$^{(B)}$ in the chain.

The algorithm takes a window Δt of n samples of a time series describing a resource utilization in the time (e.g., the CPU usage) from both the VNF$^{(A)}$ and the VNF$^{(B)}$ and computes the correlation according to the Pearson's index ρ (i.e., equation 3.1) as the covariance $\sigma_{X,Y}$ of the two variables divided by the product of their standard deviations σ_X and σ_Y. Then, it ranks the correlation by computing a discrete score, namely D-score, according to equation 3.2.

$$\rho(X,Y) = \frac{\sigma_{X,Y}}{\sigma_X \cdot \sigma_Y} \tag{3.1}$$

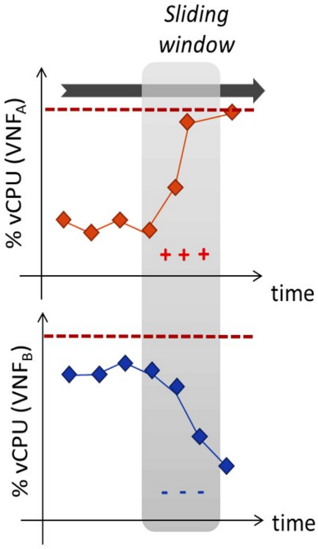


Figure 3.2. Running correlation between two VNFs

Algorithm 1: Fault correlation algorithm

```
Data: $n$: sampling window size
Data: $\Delta t$: $(t-n,\ldots,t)$ time window
Set: counter=0
begin
    foreach replica $h$ of $VNF^{(A)}$ do
        foreach replica $k$ of $VNF^{(B)}$ do
            $\rho_{h,k} = pearson(VNF^{(A)}_h(\Delta t), VNF^{(B)}_k(\Delta t))$ $\boldsymbol{D}_k = D(\rho_{h,k})$
        $\overline{D} = mean(\boldsymbol{D}_k)$
        if $\overline{D} > 0.5$ then
            $counter++$
    if $counter > |VNF^{(A)}|/2$ then
        raise alarm
```

$$D(\rho) = \begin{cases} 1, & \text{if } -1.0 \leq \rho \leq -0.7 \\ 0.75 & \text{if } -0.7 < \rho < -0.3 \\ 0.50 & \text{if } -0.3 \leq \rho \leq +0.3 \\ 0.25 & \text{if } +0.3 < \rho < +0.7 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

A zero D-score indicates a strong positive correlation between the two metrics considered, while a D-score equal to 1 indicates a strong negative correlation among them. Intermediate D-score values indicate weak correlations (i.e., $D = 0.75$, $D = 0.25$) or absence of linear correlation ($D = 0.5$). The values used in this equation are widely used in statistics to evaluate the strength of the correlation [84, 85], and do not depend on the specific system to be monitored.

Since each VNF in the chain can have multiple active replicas, the Algorithm 1 processes windows of samples gathered from each replica, and raises an alarm if more than half of the spare nodes exhibit a correlation anomaly. More precisely, for each replica h of the VNF$^{(A)}$ the algorithm computes the D-score with all the replicas k of the VNF$^{(B)}$. If the average D-score is greater than 0.5 (indicating a negative correlation) we account a possible anomaly by increasing the $counter$ variable. When all the D-scores are evaluated, if the $counter$ is greater than the half of the number of VNF$^{(A)}$ replicas, a majority of VNF instances exhibits a correlation anomaly and an alarm is raised.

The choice to wait for a feedback from a majority of nodes prevents false alarms that may be due to sporadic variations of load balancing across replicas in the same VNF stage. However, it is important to note that the algorithm is not limited to detect problems caused by multiple nodes; it is still able to detect performance anomalies caused by a single node. In the case of a single faulty node, the algorithm will detect an anomaly for the correlations between the faulty node (for example, VNF$_3^{(B)}$) and all the replicas of the previous VNF stage (for example, VNF$_1^{(A)}$ ... VNF$_n^{(A)}$) that access the faulty node. In general, the algorithm is designed to detect performance anomalies that have an impact on the capacity of a VNF stage, which can be either caused by single or multiple failures.

By executing the Algorithm 1 on sliding windows of n samples, at time t, we compute the correlation between the samples from time $t-n$, $t-(n-1)$, $t-(n-2), \ldots, t-1$. Samples are collected periodically every p seconds.

The configuration of the window, i.e., the size n and the period of sampling p, is driven by the speed at which performance anomalies are expected to happen. In our context, according to the empirical experience of industries in the ETSI consortium, performance anomalies such as avalanche restarts and overloads are expected to develop within less than 30 seconds [11]. Therefore, the n and p should be chosen such that $n \cdot p \leq 30$, as this represents a lower bound on the detection latency. For example, to have an high enough resolution to notice variations of the resource utilization metrics, and enough values to compute the correlation, the period p should be in the order of few seconds (e.g., $p = 2\text{s}$).

Of course, the need to configure an high sampling frequency may expose our approach to false alarms, that may be caused by random fluctuations of measurements. To make the approach robust to the high sampling frequency and to the choice of these parameters, we discuss two strategies to mitigate the downside of this choice:

1. use of smoothing functions on the time series to reduce the noise in the data;

2. filtering the negative correlation events according to the variance contained in the sampling window.

The first strategy requires to pre-process the sliding window with a smoothing function. Multiple algorithms can be adopted to this purpose. In Section 3.3 we compare the detection accuracy and the detection latency using three different types of smooth: (1) Running Moving Average (RMA) to lower the impact of values too distant from the average, (2) Running Moving Median (RMM) to lower the impact of values too distant from the median, and (3) Exponential Moving Average (EMA) to lower the impact of older samples in the current sampling window.

The second technique prevents spurious alarms that may occur when there is a negative correlation, but the variability of the measurements is very small and has been likely caused by random fluctuations (e.g., by chance, one of the time series may slightly increase due to random fluctuations, and at the same time the other time series may decrease). Thus, we detect a "representative" anomaly if both there is a negative correlation, *and* the variations of the measurements is large enough to reflect some event that may be occurred in the VNF (e.g., a fault or a workload change). To this purpose, we compute the *coefficient of variation* (cv) on a window of samples W, as the the ratio between its standard deviation σ_W and its mean μ_W, according to the equation 3.3. Then, a correlation between the time series is taken into account only if the cv is non-negligible, i.e., the variation exceeds the average value of the metric (typically, a coefficient of variation below 10% denotes that variations are very small [86] and could be considered random). This filter has also the advantage to exclude the sampling windows in which the chosen metric remains constant; in this specific case, the correlation index is undefined.

$$cv(W) = \frac{\sigma_W}{\mu_W} \tag{3.3}$$

Figure 3.3 shows an example of this second approach, by considering the vCPU consumption of two consecutive VNF in the pipeline. Before $t = 200s$ there are small variations in the vCPU utilization that are not representative of a change in the workload. After $t = 200s$ an increase in the load brings the VNFx in overload, while reducing the load on VNFy by 23% as a side effect (which is a consequence of resource saturation at VNFx). In correspondence of this negative correlation, there is a peak in the coefficient of variation of vCPU utilizations in both the VNFs. Thus, we consider this correlation an anomaly.

3.2 The IMS Case Study

The Clearwater IMS [46] is an open-source implementation of the IMS core standard [87]. IMS functions are implemented in software and packaged in VMs, and are designed to take full advantage of virtualization and cloud computing technology. All components can scale out horizontally using simple, stateless load-balancing based on DNS. Moreover, Clearwater follows common design patterns for scalable and reliable web services, by keeping most components largely stateless, and by storing long-lived state in clustered data stores. Clearwater is a large software project, mostly written in C++ and

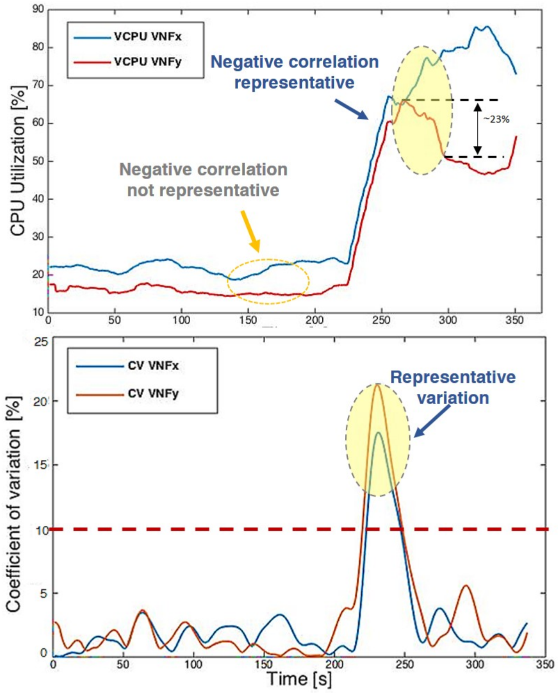


Figure 3.3. Coefficient of variation filter

Java, and including several subsystems. The architecture of Clearwater core is showed in Figure 3.4, and includes the following components:

- ***Bono*** (P-CSCF): The Bono nodes are the first point of contact for an UE (User Equipment), and they represent the edge proxy providing P-CSCF standard interfaces to IMS clients.

- ***Sprout*** (S-CSCF and TAS): The Sprout nodes are SIP registrars and authoritative routing proxies. These nodes implement the S-CSCF and I-CSCF interfaces of the IMS standard. Furthermore, they implement a distributed cache, using Memcached [88], for storing registration data and other short-lived information.

- ***Homestead***: The Homestead nodes are redundant mirrors for the HSS (Home Subscriber Server) data store, using Apache Cassandra [89], for retrieving authentication credentials and user profile information. HSS mirrors are part of both the S-CSCF and I-CSCF interfaces, and provide Web services (over HTTP) to the Sprout layer.
- ***Homer***: A Homer node is a XML Document Management Server (XDMS) to store service settings documents for each user of the system, using Apache Cassandra as the data store.
- ***Ralf*** (Rf-CTF): The Ralf nodes provide charging and billing functions, used by Bono, Sprout and Homestead nodes to report events occurring when the CSCF chain is traversed.

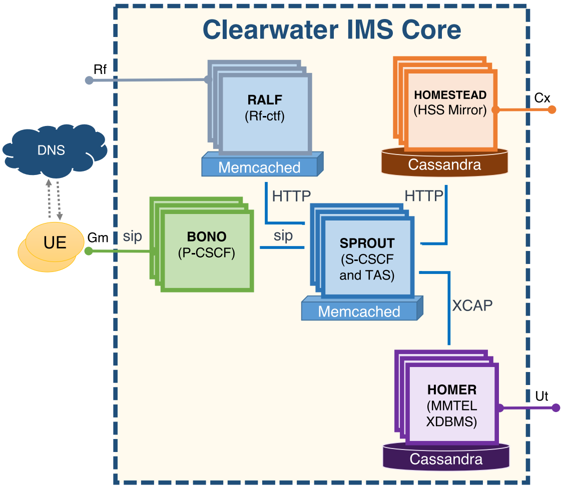


Figure 3.4. Architecture of the Clearwater IMS.

The experimental testbed (Figure 3.5) consists of four host machines: three Dell PowerEdge R520 servers, equipped with two 8-Core 2.2 GHz Intel Xeon CPU, 64GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber Channel HBA; one Dell PowerEdge R320 server with a 4-Core 2.8 GHz Intel Xeon CPU, 8GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber Channel HBA; A PowerVault MD3620F disk array with 4TB of network storage with a 8-Gbps Fiber Channel link.

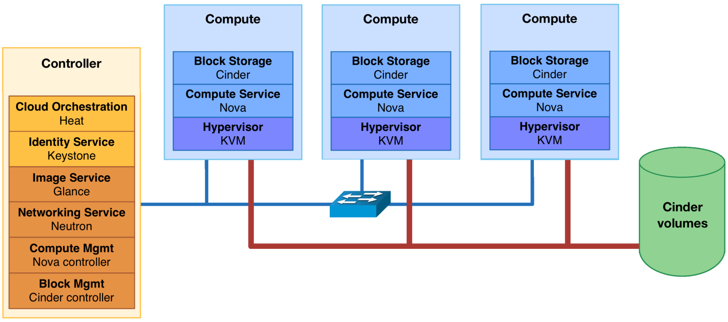


Figure 3.5. Experimental testbed.

The hosts are connected to a 1-Gbps Ethernet network for general-purpose traffic, and another 1-Gbps Ethernet network for management traffic. The virtual disks of VMs are stored on three distinct GlusterFS partitions of the PowerVault SAN, which are mounted on the hosts through the Fiber Channel link.

The hosts are configured with CentOS Linux 7 and the KVM hypervisor. The testbed is managed using the *OpenStack* virtualization platform, version Juno [90]. The Dell PowerEdge R320 serves as OpenStack Controller and Network node; the three Dell PowerEdge R520 servers represent the OpenStack Compute and Storage nodes, and run the VMs of the Clearwater IMS. The OpenStack services include: *Nova*, which manages the compute domain; *Neutron*, which manages virtual networks among VMs; *Cinder*, which controls the lifecycle of VM volumes; *Glance*, which manages the cloud images of VMs; *Heat*, which orchestrates, through a native REST API, the virtual IMS deployment; *Horizon*, which supports the Web-based management dashboard.

To determine the number of VMs that had to host specific network ser-

Table 3.1. Clearwater VMs deployment configuration.

Service	**Clearwater Node Name**	**# of VMs**	**Flavor Details**
Edge Proxy (P-CSCF)	Bono	10	VCPUs: 1 RAM: 2GB Disk Size: 5GB
SIP Router (I/S-CSCF)	Sprout	10	VCPUs: 1 RAM: 2GB Disk Size: 5GB
HSS Mirror	Homestead	5	VCPUs: 1 RAM: 4GB Disk Size: 80GB
Rf CTF	Ralf	4	VCPUs: 1 RAM: 2GB Disk Size: 5GB
XDMS (MMtel services)	Homer	2	VCPUs: 1 RAM: 4GB Disk Size: 100GB
Name service (DNS)	-	1	VCPUs: 1 RAM: 2GB Disk Size: 5GB
Workload generator (SIPp)	-	10-40	VCPUs: 1 RAM: 2GB Disk Size: 100GB

vices, we made some preliminary capacity tests. We defined a deployment configuration capable to handle *500,000 subscribers* (i.e. **the engineered capacity**) corresponding to (i) 90,000 registration attempts per minute, and (ii) 8,000 call attempts per minute. At this level the average CPU utilization is 80% in all the Clearwater VMs and all the requests are correctly served by the system. The number and type of VMs hosting services is detailed in Table 3.1. Each VM hosts a single VNF.

Other VMs are used to generate the IMS workload. Such machines run the **SIPp traffic generator**. Each SIPp instance generates SIP traffic towards a specific P-CSCF instance. Each couple of subscribers will attempt to register or renew the registration every 5 minutes, on average. After a successful registration, one can attempt to setup a call to the other (with 16% of probability) or remain idle until the next registration renewal (with 84% of probability). The call hold time is, by default, 60 seconds. 10 SIPp are used for generating the initial load of 500,000 subscribers in 10 minutes (*Initial Ramp-up period*). To generate the overload conditions in our test scenarios, we run 40 additional SIPp VMs.

3.3 Experimental evaluation

In our experimentation, we study the ability of the detection algorithm to identify performance anomalies, and to avoid false positives. In the context of the IMS case study, such anomalies cause the failure of some user registrations and/or some call setup requests. On the opposite, when there are no faults affecting the quality of service, all the registrations and the call setups are correctly processed by the system. Thus, we use the SIPp workload generator to check at client-side the success of such requests, in order to evaluate the outcome of the detection algorithm.

In our evaluation, we consider test scenarios that involve service failures of the IMS system. To have meaningful test scenarios, we induce performance anomalies that cause service failures, that is, the quality of service experienced by clients degrades, either in terms of throughput (i.e., there should be a gap between the request rate from the client, and the throughput of traffic served by the IMS) and latency (i.e., there is a long delay between a request and the corresponding results). In quantitative terms, we cause service failures where the throughput is less than 90% of the request rate for more than 5 seconds, and the 90th-percentile of the request latency is lower than 250ms. The requests that violate the latency requirements are signalled either by the system (i.e., with SIP 500 messages) or by the client (i.e., in case of timeout events). In both cases, these requests are marked as failed and are not accounted in the overall throughput. Thus, in our discussion, we focus on presenting the throughput metric, as in all tests the latency violations were always accompanied by throughput violations during the same periods.

To assess the detection algorithm, we consider a set of overload scenarios (caused by workload surges and faults), and perform r repeated experiments for each scenario, where we evaluate the number of times the algorithm is able to detect the overload. The following *Detection Outcome*s are considered:

- *Overload not detected*: the algorithm detected the overload no more than in 20% of the experiments;

- *Overload detected*: in at least 80% of the experiments, the algorithm was able to detect the overload;

- *Unreliable detection*: in the other cases.

To summarize the detection outcomes across different scenarios, we compute the *Overall Detection Coverage*, which we define as the percentage of the scenarios where the detection outcome is *overload detected*.

Another requirement of NFV services is that anomalous conditions have to be detected as soon as possible, so that mitigation mechanisms can be quickly activated, and the impact on the quality of service can be reduced. Thus, as a further metric for the assessment of the detection algorithm, we consider the *Detection Latency*, which is defined as the time between the occurrence of an overload condition (i.e., the moment at which users' registrations and/or calls start failing) and the detection of such condition by the algorithm.

Finally, we consider the rate of false alarms that are raised by the detection algorithm. To this aim, we perform experiments without anomalies, and keep track of any (false) alarms raised by the algorithm during the experiment.

Ideally, to be deployed in production environments according to the feedback from our industrial partners, our proposed algorithm should have a quick detection latency and no false positives, and a reasonably high detection coverage; this can be a challenging goal considering that we do not rely on any preliminary calibration of thresholds (e.g., we do not fix a minimum or maximum value for CPU or bandwidth utilization in our algorithm).

We applied the proposed approach by correlating the **CPU utilization** of VNFs in the service chain. One of the reasons why we focus on this metric is that, in NFV services, the network consumption is highly correlated to the CPU utilization, since NFV is intended to use standard COTS CPUs to process high volumes of network traffic.

Figure 3.6 shows an example of correlation, in the presence of a performance anomaly, between the first two components of the Clearwater VNF chain, the *P-CSCF CPU %* (Bono) and *S-CSCF CPU %* (Sprout). The figure shows the time series for vCPU utilization of two instances of these network functions, and the Pearson correlation index (the yellow line) computed between these two, by using a sliding window. A workload surge is generated at minute 10. After minute 10, the Bono node starts dropping new connection attempts due to the overload, thus causing a reduced load on the subsequent Sprout node. When this happens, the correlation index drops close to -1, and our algorithm considers this as a symptom of fault (a performance anomaly). Analogue conditions occur in all the other failure scenarios that we consider in the experimental evaluation.

In summary, we consider the following sets of experiments:

1. **Sudden workload surges**: the workload of the system rapidly grows, exceeding the engineered level of the IMS. In such a case, the available resources of the system may not suffice to manage the incoming load.

2. **Component failure**: the failure of a component of the system reduces

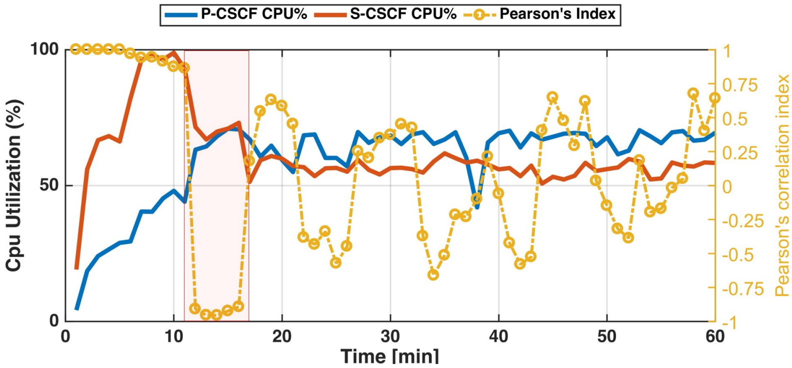


Figure 3.6. Example of negative running correlation between P-CSCF and S-CSCF CPU utilization.

available resources to satisfy all user requests, thus, causing an overload condition.

3. **Anomaly-free, long-running workload**: we consider long-running tests, with both constant and variable workloads, within the engineered level of the IMS and without any fault, to check whether any normal variation of the workload may trigger false positives.

In each scenario, we apply the fault correlation approach to the main service chain of the Clearwater IMS, including Bono, Sprout and Homestead, as shown in Figure 3.7. More precisely, we apply the fault correlation algorithm to both the *Bono-Sprout* and *Sprout-Homestead* VNF pairs. We do not consider the *Sprout-Ralf* VNF pair since the external billing function (required by Ralf) is not included in Clearwater. Moreover, we do not consider homer, since it only provides a secondary functionality (a database service for the Telephony Application Server) that is not included in the IMS standard.

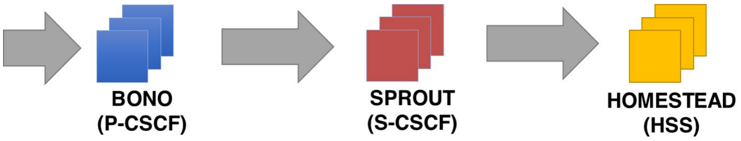


Figure 3.7. VNF graph representing the chain of services' utilization.

3.3.1 Sudden workload surges

We study the impact of different types of workload surges on the QoS of the IMS and on the effectiveness of our detection approach. In each test with workload surges, we consider a different combination of the following three factors:

- The *number of subscribers*, as the user volume affects the severity of resource contention and of the saturation of the IMS capacity;
- The duration of the *ramp-up period*, that is, the time for the workload to increase from the engineered level to the selected level (the shorter is the ramp-up, the quicker is the workload surge and the on-set of the overload condition);
- The *call hold time*, which affects the the type and frequency of requests to the IMS, and consequently lead to different workload patterns.

Table 3.2 reports in the detail the possible values that we selected for the three factors (four possible numbers of subscribers, three possible ramp-up periods, and two possible call hold times). In particular, the number of subscribers is expressed in relative terms with respect to the engineered level, that is, the users are 20, 100, 640, 1000% more numerous than normal (denoted with *X%-MTN*). We adopted a full factorial design, with $4 \times 3 \times 2 = 24$ test configurations in total. In these experiments, workload surges are introduced

Table 3.2. Factors and levels for studying the impact of workload surges.

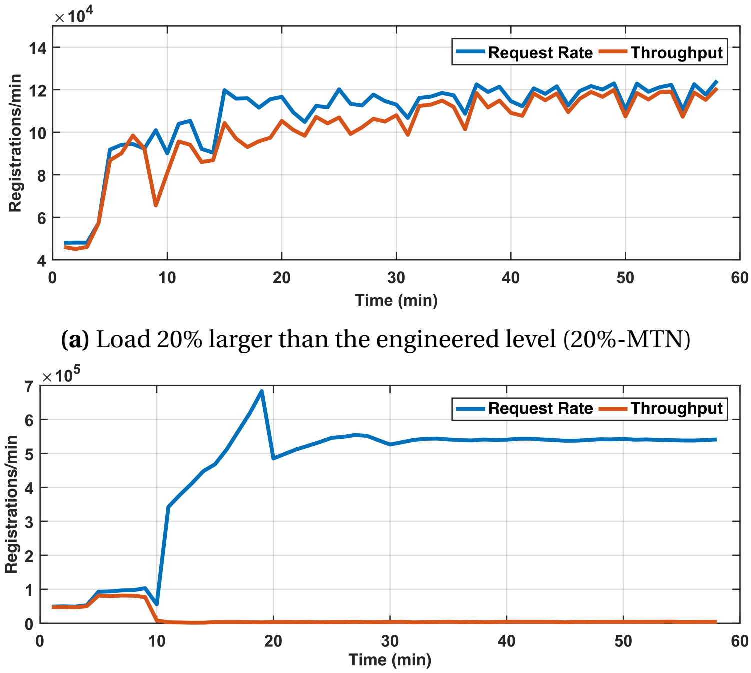
Factor	Level 1	Level 2	Level 3	Level 4
# subscribers	600k 20%-MTN	1M 100%-MTN	3.2M 640%-MTN	5.5M 1000%-MTN
Ramp-up	10 min	6 min	3 min	
Call hold time	2 min	1 min		

starting at minute 10 since the beginning of the experiment; the time required to reach the peak of subscribers depends on the ramp-up period.

We found that covering boundary conditions (e.g., relatively high and relatively low volumes of users) highlights different behaviors of the IMS: in these extreme cases, either just few registrations and calls fail (but have still a noticeable effect on the perceived QoS), or almost all registrations and calls fail (as the resource competition is too strong to allow any request to get a sufficient amount). These differences also reflected on the performance of the detection algorithm. Instead, we found that the ramp-up period and the call hold time have a limited influence on the performance of detection; thus, we present detailed results only to a specific ramp-up period (i.e. 10 minutes) and call hold time (i.e. 60 seconds).

We performed 5 repeated experiments for each test configuration ($r = 5$). A performance anomaly condition occurs when a non-negligible percentage (≥10%) of user registrations and/or call setups are not successful, either because the request is not served within a time limit (10 seconds), or the IMS explicitly refuses the request and returns an error message to the client. The anomaly condition is considered detected if the algorithm raises an alarm within 60 seconds from the occurrence of registration and/or call failures.

Figure 3.8 presents two examples of overload due to the increase of the number of subscribers. Figure 3.8a shows, respectively, the number of *incoming* registration requests per minute, and the number of *completed* registra-



(a) Load 20% larger than the engineered level (20%-MTN)

(b) Load 1000% larger than the engineered level (1000%-MTN)

Figure 3.8. Registration attempts per minute and registrations completed per minute

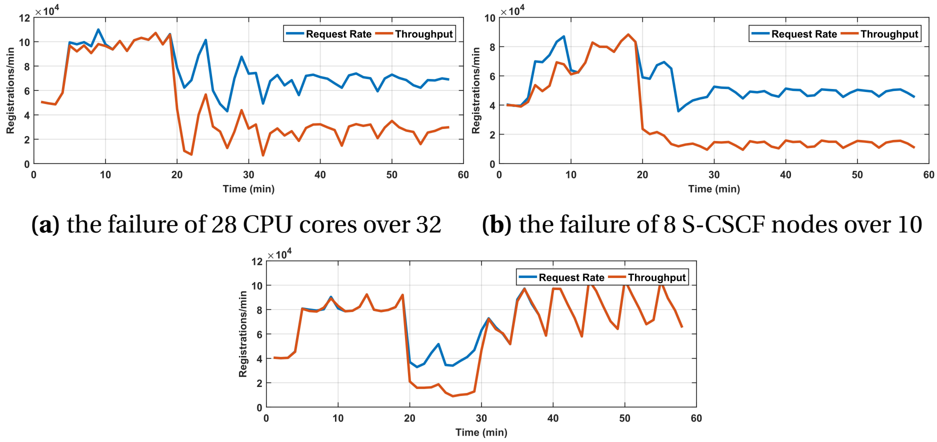
tions per minute, when the workload is 20% larger than the nominal capacity. The difference between the two curves represents the amount of requests that could not be service due to resource contention and saturation. Similarly, Figure 3.8b shows the case where the number of subscribers increases by 1000%. In the former case (20%-MTN), the workload peak affected the quality of service for a small share of users, while the others were still serviced. Instead, in the latter case (1000%-MTN), not only the users in excess could not be serviced; but the workload surge caused a failure of the IMS software (which was unable to allocate resources, such as memory), thus leading to the unavailability of the IMS. Clearly, the larger the increase of the number of subscribers, the larger the number of registrations that are not correctly completed.

To evaluate the detection algorithm based on the running correlation, we consider several values of sample window size, i.e., we vary the number of samples from the time series that are correlated. Also, we evaluate detection performance when using different smoothing algorithms. Specifically, we consider to use (i) 10, (ii) 20 or (iii) 30 samples; and, as a smoothing algorithm, we test (i) Running Moving Median (RMM), (ii) Running Moving Average (RMA), and (iii) Exponential Moving Average (EMA).

The results for the detection algorithm under workload surges are reported in Table 3.3. Clearly, the size of the sampling window and the smoothing function have a big impact on the detection performance. In all the considered overload conditions, the RMM and RMA smoothing functions perform better than EMA. This result is probably due to the fact that giving less importance to older samples in EMA, makes the algorithm more sensitive to noisy peeks revealing trends that are not representative. Moreover, the RMM algorithm appears more robust than RMA regarding the size of the sampling window due to the fact that the mean is more sensitive to outliers than the median. This results in lower detection latencies. In general, the average detection latency varies between 30 and 60 seconds and increases when using bigger sampling windows. Collecting a sample every 2 seconds, a sampling window of 10 samples requires at least 20 seconds to be filled. Longer windows (e.g, 30 samples) result in worst coverage and longer detection latencies, especially with small overload conditions. For this reasons we recommend to use small sampling windows and the more robust RMM smoothing algorithm to achieve good results. With this configuration, we obtain 100% of detection coverage and an average detection latency of 32 seconds.

Table 3.3. Detection outcomes and latency under workload surges.

Overload Subs. (MTN)	**Window**	**Smooth**	**Detection Outcome**	**Detection Latency (seconds)**
20%	10	RMM	Detected (4/5)	29.0
	20		Detected (4/5)	45.6
	30		Not Det. (1/5)	28.0
	10	RMA	Unrel. Det. (2/5)	37.0
	20		Non Det. (1/5)	48.0
	30		Not Det. (0/5)	-
	10	EMA	Unrel. Det. (3/5)	46.0
	20		Non Det. (0/5)	-
	30		Not Det. (0/5)	-
100%	10	RMM	Detected (4/5)	29.0
	20		Detected (4/5)	44.0
	30		Unrel. Det. (2/5)	57.0
	10	RMA	Detected (4/5)	33.2
	20		Detected (4/5)	47.2
	30		Not Det. (0/5)	-
	10	EMA	Detected (4/5)	36.5
	20		Detected (4/5)	42.0
	30		Not Det. (0/5)	-
640%	10	RMM	Detected (5/5)	42.4
	20		Detected (5/5)	58.0
	30		Detected (5/5)	46.2
	10	RMA	Detected (5/5)	49.3
	20		Non Det. (1/5)	58.0
	30		Non Det. (0/5)	-
	10	EMA	Detected (5/5)	46.0
	20		Non Det. (0/5)	-
	30		Non Det. (0/5)	-
1000%	10	RMM	Detected (5/5)	29.4
	20		Detected (5/5)	49.4
	30		Detected (5/5)	46.2
	10	RMA	Detected (5/5)	38.0
	20		Detected (4/5)	57.0
	30		Non Det. (0/5)	-
	10	EMA	Detected (5/5)	36.0
	20		Non Det. (2/5)	57.0
	30		Non Det. (0/5)	-



(a) the failure of 28 CPU cores over 32

(b) the failure of 8 S-CSCF nodes over 10

(c) the failover of 8 P-CSCF nodes over 10

Figure 3.9. Registration attempts and registrations completed per minute, under component failures (due to faults injected at minute 20).

3.3.2 Component failure

We here analyze how component failure inside the NFV infrastructure (and thus, the variation of the capacity of the service chain) impacts on the QoS and on the effectiveness of the detection algorithm. We consider the following potential failure events:

1. The failure of physical CPU cores of a machine that hosts VNFs, which is emulated by deliberately turning off a subset of CPU cores, thus forcing the hypervisor and the VNFs to run on fewer CPU cores and causing physical CPU contention.

2. The crash of VMs that run VNF software, which is emulated by deliberately terminating a VM, thus forcing the IMS traffic to be load-balanced on the remaining replicas of the VNF.

3. The restart of VMs that triggers the migration and restart of IMS sessions. In the telecom domain, this phenomenon is often referred to

as the *avalanche effect*, and is regarded as a problematic event due to the need to quickly restart a high number of connections in a limited time, and to force state migration in the case of stateful network functions [11].

Table 3.4 reports in the detail the levels for experimenting with component failures. In total, we consider 4 test configurations, with $r = 5$ repetitions for each configuration.

Table 3.4. Factors and levels for studying the impact of failure events.

Factor	**Level 1**	**Level 2**	**Level 3**	**Level 4**
Failure	16 out of 32 pCPU failure	28 out of 32 pCPU failure	8 out of 10 S-CSCF failure	8 out of 10 P-CSCF failover

We apply the first type of failure on one of the three physical nodes that run the IMS; the second type of failure on S-CSCF services, by killing 8 VMs running the Clearwater Sprout service; and the third type of failure on P-CSCF services, by restarting 8 VMs running Bono, thus triggering the P-CSCF recovery. All the injections are performed 20 minutes after the start of the experiment.

Figure 3.9 shows examples of the impact caused by the failures on the IMS. For three out of four failure events (the levels 2, 3, and 4 in Table 3.4), the injected faults indeed caused an overload of the IMS system, since many users were affected by failures due to unsuccessful registrations. Instead, in the remaining case (the level 1 in Table 3.4), the CPU failure were not enough to cause an overload condition, as the IMS client did not perceive any service degradation. The IMS components tolerate small a amount of physical CPU contention (e.g., the loss of 10% of CPU time, spent in involuntary wait state) with no effects on the throughput and the latency. Therefore, we decided to consider this experiment as anomaly-free (the remaining CPUs were able to tolerate the component failure and to serve the workload, so no anomaly should be detected for this case). This case is further analyzed in the next section.

To analyze the detection algorithm under component failures, we focus the discussion on the case with a workload below the engineered capacity (i.e. 400k subscribers), sampling window size of 10 samples and a sampling period of 2s (i.e., the window length is equal to 20s), and we apply the Running Moving Median (RMM) as smoothing function; these choices for the window size and smoothing were the best ones according to the previous analysis with workload surges. Again, we have a performance anomaly when a noticeable amount of user registrations and/or call setups are not successful. The anomaly is considered detected if the algorithm raises an alarm within 60 seconds from the occurrence of registration and/or call failures.

Results from these injection experiments, reported in Table 3.5, reveal a high detection coverage. The mean detection latency (i.e., 18 seconds) is approximately equal to the time required to fill the window (i.e., 20 seconds) with samples collected after the injected fault. The detection of this kind of issues is faster than the case with workload surges, because the injection of the faults caused quicker variations of the CPU utilization in a majority of the VMs, all at the same time. In the case of CPU contention (e.g., caused by the CPU failures), all the VMs deployed on the same injection target experienced involuntary waits due to the hypervisor scheduler. In the case of a reduced number of VMs (e.g., due to the crash of S-CSCF instances) the algorithm required less feedback to reach the majority before raising an alarm, resulting in lower detection latency. In case of avalanche restarts (e.g., due to the failover of P-CSCF nodes) all the newly started instances immediately experienced overload and the algorithm got a quick feedback from a majority of the nodes.

Table 3.5. Results for detection based on running correlation for overload conditions due to failures.

Failures	**Window**	**Smooth**	**Detection Outcome**	**Detection Latency (seconds)**
physical CPU contention	10	RMM	Detected (4/5)	13.0
S-CSCF crash	10	RMM	Detected (5/5)	24.0
P-CSCF failover	10	RMM	Detected (5/5)	18.0

3.3.3 Anomaly-free, long-running workload

To test for the occurrence of any false alarms under anomaly-free conditions, we carry out a set of experiments that are within the engineered capacity of the IMS system. We consider both the case of a stable workload at the engineered capacity, and two scenarios with variable workload (still within the limits of the engineered capacity). Finally, we consider the case of a failure event that reduces the available physical CPU cores (16-out-of-32) while still providing enough capacity for serving the workload (see also the discussion in the previous section). In these conditions, the algorithm should not detect any failure, thus any alarm is considered a false positive.

In the case of stable workload, we exercise the IMS with a constant number of subscribers (500k users). In the case of variable workload, we vary the number of subscribers over time. Periodically (every 20 minutes on average) the number of subscribers is reduced or increased, according to two patterns: in the first pattern (Figure 3.10) the workload varies between three levels below the engineered capacity; in the second pattern (Figure 3.11), the workload varies between five levels, up to the engineered capacity of the IMS.

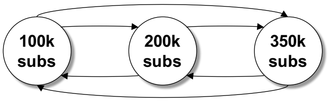


Figure 3.10. Variable workload below the engineered capacity.

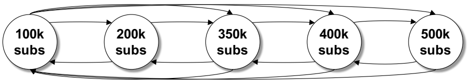


Figure 3.11. Variable workload that saturates the engineered capacity.

In all these experiments, the detector provided an encouraging result: no false alarms were raised, for all test configurations. This result is motivated by the robust criteria that we adopt in the algorithm, as we require that (i) the CPU utilization should not simply vary on individual nodes, but the variations should be correlated at different pairs of VNFs; (ii) the correlation should show a high strength; (iii) a majority of the replicas in a VNF tier should be involved in the variation. Indeed, it is very unlikely that a false positive may occur, as confirmed by our anomaly-free experiments.

Chapter 4

Managing the overload of network functions in the Cloud

Network Function Virtualization (NFV) aims to provide high-performance network services through cloud computing and virtualization technologies. However, *network overloads* represent a major challenge. While elastic cloud computing can partially address overloads by scaling on-demand, this mechanism is not quick enough to meet the strict high-availability requirements of “carrier-grade” telecom services. Thus, this Chapter presents a novel overload control framework (*NFV-Throttle*) to protect NFV services within a short period of time, by filtering the incoming traffic towards VNFs in order to make the best use of the available capacity, and to preserve the QoS of traffic flows admitted in the network. Moreover, the framework has been designed to fit the service models of NFV, including *VNFaaS* and *NFVIaaS*. Moreover, this chapter presents an extensive experimental evaluation on the NFV-oriented *Clearwater* IMS, showing that the solution is robust and able to sustain severe overload conditions with a very small performance overhead.

4.1 The problem of overload control in NFV

When a network element becomes overloaded, a large amount of traffic can be lost very quickly. The overload may cause the disruption of already-established connections and the unavailability of high-priority services, thus violating SLAs. Moreover, overloads may expose network services to cascading failures due to user retries, traffic handover, and avalanche restarts. The objective of overload control is to guarantee that the network is still able to serve a high number of traffic flows by fully utilizing its capacity, and to assure that an adequate QoS (for example, in terms of latency, packet loss, and "goodput") is provided to flows that are admitted in the network.

This problem is exemplified in Figure 4.1. Typically, the network capacity is designed according to technical and economical considerations, in order to support some "Reference load" (point C1), for example in terms of amount of traffic per second. Under this load level, the network can perform well, and assures an "engineered throughput". However, when a mass event or a cascade failure occurs, the network becomes overloaded ("Overload condition", point C2 in the figure). The network does not have enough resources to process all the incoming flows. Thus, if the overload condition is not managed, the network throughput can significantly degrade (dashed curve in the figure). Ideally, using overload control, the network should maintain a steady throughput (for example, no lower than 90% of the engineered throughput, the continuous curve in the figure) even under an overload condition, by dropping or rejecting the traffic in excess, in order to accept only few traffic flows in the network, and by efficiently using its resources.

According to this view, the NFV overload control solution should consider the following requirements:

1. *The NFV network should achieve an acceptable level of service (for example, not less than 90% of its engineered throughput) during severe overload*

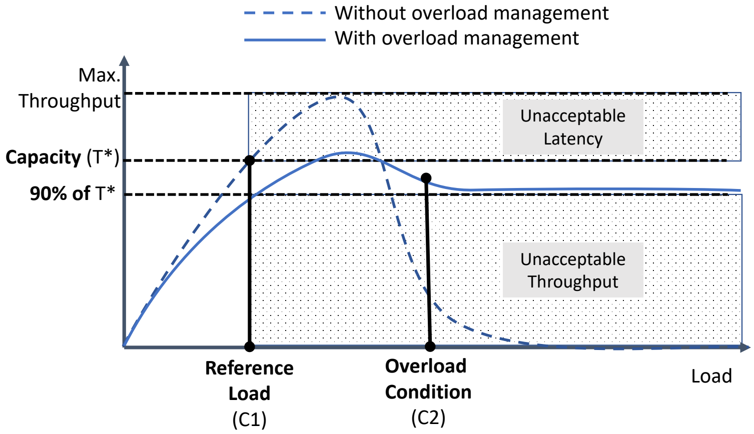


Figure 4.1. Network throughput under overload conditions.

conditions (such as 10 times the reference load).

2. *The overload control solution should quickly react to an overload condition, in order to prevent violations of SLAs during the transition between a normal load and the overload condition. Since carrier-grade services can afford only few minutes of downtime per year, it is important to react to overloads within few tens of seconds at most.*

3. *The overload control solution should be integrated with the use cases and scenarios of NFV, including VNF providers, and NFVI providers. For VNF providers, it is desirable that the solution is transparent to VNF software, which can be developed by third-party vendors and whose source code may not be available. Moreover, the solution should allow NFVI providers to perform overload control at the infrastructure-level, without relying on cooperation of the VNF layer.*

4. *The overload control solution should introduce minimal overhead, and must not degrade the quality of service under normal load conditions (for example, it should not filter traffic when processing resources are available).*

4.2 The proposed overload control solution

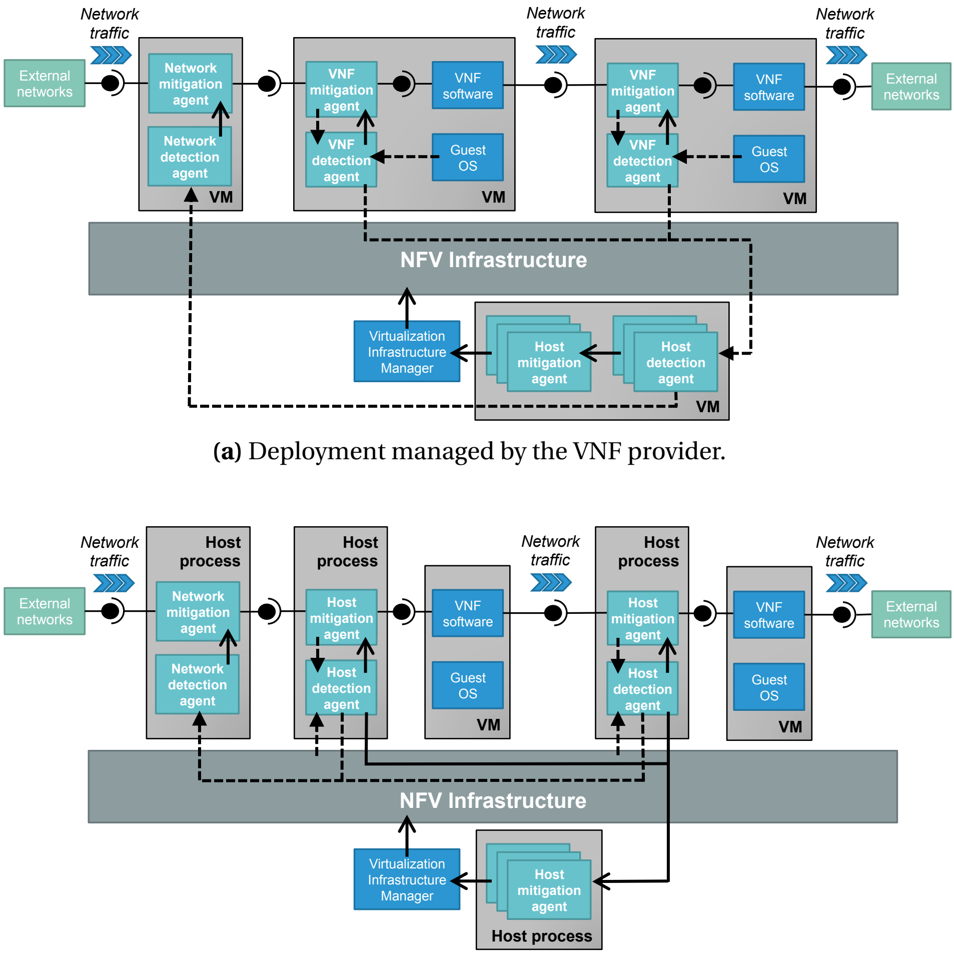
In the following, I describe an overload control solution aimed at fulfilling high-availability and performance requirements of Telecom services, and at complying with the service models of NFV and cloud computing. In particular, we consider two main use-case scenarios:

1. A telecom operator designs a network service (e.g., to offer it as a service, *VNFaaS*), by assembling VNFs and composing them into a VNF service chain (see Figure 4.2a). The VNFs can run VNF software developed in-house or provided by third-party NFV software vendors. The VNFs are deployed on an NFVI managed by a third-party NFVI provider (*NFVI-aaS*). In this scenario, the telecom operator can customize the VNFs and deploy VMs on the NFVI, but it cannot change the underlying NFVI.

2. An NFVI provider manages an infrastructure (e.g., to offer it as a service, *NFVIaaS*) to host VNFs from telecom operators (see Figure 4.2b). In this scenario, it is desirable (or even mandatory, if the VNFs are provided as black-boxes) to address overloads at the infrastructure level, without making changes to VMs.

The proposed solution is an overload control framework based on a set of *overload detection agents* and *overload mitigation agents*. These agents are software modules to be deployed inside the NFV network, and transparent to VNF software (Figure 4.2):

- **Overload detection agents** check whether the incoming traffic towards the VNF exceeds its capacity, either due to a workload peak, or due to contention on physical resources of the NFVI. If an overload condition occurs, the detection agent triggers an overload mitigation agent.



(a) Deployment managed by the VNF provider.

(b) Deployment managed by the NFVI provider.

Figure 4.2. Overview of the overload control solution.

- **Overload mitigation agents** protect the VNF from incoming traffic in excess, by dropping it, or by only admitting a subset of users to the service, and it allows again the traffic once the overload condition disappears. Moreover, overload mitigation agent interact with the virtualization infrastructure manager (VIM) to reserve physical resources to critical VNFs, in order to mitigate physical resource contention.

Overload detection and mitigation agents are further divided in three complementary types. The *VNF-level agents* protect individual VNFs, and react to overload conditions by dropping traffic in excess. The *host-level agents* protect groups of VNFs that share the same physical host, with respect to overload conditions that arise from physical resource contention. Finally, *network-level agents* protect the NFV network from overload condition that affect the whole network (i.e., overloads spread across several VNFs), and react by rejecting traffic and notifying the clients about the overload condition.

The agents can be deployed across the NFV network to support any of the two use-case scenarios mentioned before:

1. A telecom operator can install the detection and mitigation agents both in the same VMs of VNFs, and in dedicated VMs (Figure 4.2a). The **VNF-level** agents collect resource utilization metrics from VMs, and forward traffic to VNF software through a transparent network tunnel, which drops traffic in excess in the case of an overload. Moreover, the telecom operator can deploy **host-level** agents on dedicated VMs, which detect physical resource contention impacting on the VNFs, and mitigate it by re-configuring VMs. Finally, **network-level** agents are deployed on dedicated VMs, and are interposed between the NFV network and external networks. They detect overload conditions that are spread across several VNFs, and use a transparent network tunnel in order to forward network traffic and to reject traffic in excess.

2. An NFVI provider may not be allowed to install agents inside the VMs of VNFs, but has the opportunity to install agents in the physical hosts of the NFV infrastructure (Figure 4.2b). **Host-level** detection and mitigation agents are deployed as processes or services running on the

physical hosts. The host-level agents use a transparent network tunnel towards each VNF, by leveraging virtual networking mechanisms provided by the infrastructure, in order to protect an overloaded VNF from ingoing traffic in excess. In a similar way, **network-level** agents can be deployed on physical hosts and can be interposed between the NFV network and external networks. Moreover, host-level agents can be adopted to mitigate physical resource contention on the host.

The proposed overload control framework is designed to react to overload in the short term (e.g., few tens of seconds), and is complementary to elastic cloud computing mechanisms that expand the capacity of VNFs. The framework does not require to change VNF software and virtualization software, and can be transparently installed into NFV networks with third-party VNF software and virtualization technologies. The overload detection agents only rely on metrics that are widespread across guest OSs and hypervisors, and that are easily collectible through APIs or IPC channels exposed by the guest OSs and hypervisors, without modifying their internals. Moreover, the solution gives to NFV designers and administrators the ability to install agents only for specific VNFs, where overload control is most needed; reuse the overload control framework across different types of network functions; to address overload either at VNF- or at host-level, and/or globally at the NFV network level.

4.2.1 VNF-level design

The architecture of the overload control solution at VNF level is showed in Figure 4.3, which includes a detection agent and a mitigation agent.

VNF-level Detection Agent

The VNF-level Detection Agent is a component deployed by a VNF provider inside a VM, in order to address overloads of an individual VNF (Figure 4.2a).

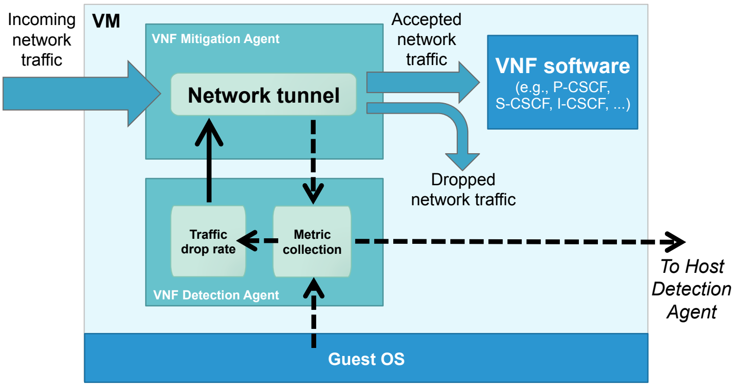


Figure 4.3. Architecture of VNF-level detection and mitigation.

It collects resource utilization metrics from the VM, by using interfaces exposed by the guest OS (such as the *procfs* virtual filesystem of the Linux OS). Specifically, it collects metrics about the utilization of virtual CPU by the VM. These metrics include the *busy* virtual CPU ticks, consumed both by user-space applications (including VNF software) and by the guest OS (including system calls and interrupt service), and the *idle* CPU ticks of the VM. Moreover, to relate overload conditions to the workload of the VNF, the VNF-level Detection Agent measures the ingoing and outgoing traffic throughput of the VNF. Network traffic metrics are collected from the VNF-level Mitigation Agent (discussed later in this section), which tunnels network traffic to the VNF software.

The VNF-level Detection Agent must quickly react to an overload condition within a short time frame (e.g., 10 seconds for critical NFV networks). Therefore, the VNF-level Detection Agent periodically samples resource utilization metrics, and continuously updates a *traffic drop rate* in near real-time, using a simple and robust update rule, which is defined as:

$$\text{capacity} = \frac{\textsc{Mean}[\text{accepted\_traffic}[1\ldots N]]}{\frac{\textsc{Max}[\text{cpu\_usage}[1\ldots N]]}{\text{reference\_cpu\_usage}}} \tag{4.1}$$

$$\text{drop\_rate} = 100 \cdot \left(1 - \frac{\text{capacity}}{\text{incoming\_traffic}[N]}\right) [\%] \tag{4.2}$$

where *cpu_usage* is a sliding window of the latest N samples of the percentage of busy virtual CPU ticks (up to 100%); *incoming_traffic* is the volume of traffic in input to the VNF-level Mitigation Agent; and the *accepted_traffic* is the volume of traffic that is actually passed to VNF software by the agent. In these equations, $\textsc{Max}[\text{cpu\_usage}[1 \ldots N]] > 0$, and $\text{incoming\_traffic}[N] > \text{capacity} > 0$; otherwise, if $\text{capacity} > \text{incoming\_traffic}[N]$, then the traffic drop rate is set to zero. The traffic drop rate is capped between 0% and 100%.

The *drop_rate* for VNF traffic is updated following Alg. 2. The VNF-level Detection Agent periodically collects a new sample of resource utilization metrics (*cpu_usage*, *incoming_traffic*, and *accepted_traffic*) at a high frequency (later in this study, we configure the agent to collect one sample every 2 seconds). Moreover, the VNF-level Detection Agent analyzes the most recent N samples (e.g., we consider the last $N = 5$ samples when sampling every 2 seconds) of virtual CPU utilization and of the network traffic throughput. The VNF-level Detection Agent first identifies the highest virtual CPU utilization sample among the recent samples, and compares it to a reference virtual CPU utilization. The reference virtual CPU utilization is chosen by NFV designers or administrators: it represents a "factor of safety" for virtual CPU utilization, under which the VNF is designed to perform well (e.g., no service disruptions), as discussed in section 4.1. For example, the VNF software and configuration can be designed (e.g., through capacity planning of virtual and physical resources) to have a virtual CPU utilization below 90% and to provide good performance under a reference workload.

If the virtual CPU utilization exceeds the reference virtual CPU utilization, the traffic allowed into the VNF (*capacity*) is reduced by the update rule (eq. (4.1)). The new value is obtained by scaling down the average of the most recent N samples of traffic volume accepted into the VNF; the scaling is proportional to the gap between the reference virtual CPU utilization and the actual virtual CPU utilization. Thus, the larger the gap, the lower the capacity, and the higher the traffic drop rate.

This computation is periodically repeated for each new sample of resource utilization. If the overload condition persists (i.e., the CPU utilization is still higher than the reference value), the *accepted_traffic* and the *capacity* will keep reducing, and the *drop_rate* will further increase. Instead, when the VNF leaves the overload condition (i.e., the virtual CPU utilization is below the reference value), the VNF-level Detection Agent will gradually increase the capacity and reduce the traffic drop rate, until it becomes zero (that is, all the input network traffic is again allowed in the VNF software). At each update, the traffic drop rate is sent to the VNF-level Mitigation Agent. Finally, the VNF-level Detection Agent sends periodic updates on virtual CPU utilization to the Host-level Detection Agent, in order to detect physical resource contention, as discussed later in this section.

This approach is robust to false positives, since a sporadic increase of the virtual CPU (e.g., transient peaks in the samples that are not due to an overload condition, but are due to random effects) is quickly discarded since we adopt a relatively small window of samples (e.g., $N = 5$), which only causes to drop a small amount of traffic and a negligible impact on the quality of service. In the case of a larger window, the update rule can be changed by replacing the $\textsc{Max}[\cdot]$ function with a percentile (such as the 90^{th} percentile among the N samples). Moreover, the VNF-level Detection Agent applies a moving-average filter to the samples of network traffic throughput, which lessens the effect of sporadic out-of-norm samples from network measurements.

VNF-level Mitigation Agent

The VNF-level Mitigation Agent acts as a network tunnel between the VNF software and other VNFs in the NFV network. The network traffic towards

Algorithm 2: VNF-level detection and mitigation

```
Data: SP: sampling period
Data: N: size of the vector of samples
Data: reference_cpu_usage: "factor of safety" for virtual CPU usage
Result: drop_rate for incoming VNF traffic
begin
    while True do
        collect cpu_ticks, incoming_traffic and accepted_traffic
          measurements;
        update capacity and drop_rate;
        send cpu_ticks to the Host-level Detection Agent;
        send the updated drop_rate to VNF-level Mitigation Agent;
        wait SP seconds;
```

the VNF software is forwarded to the VNF-level Mitigation Agent. In turn, the VNF-level Mitigation Agent connects to the VNF software, and it forwards the traffic to the VNF software.

This forwarding is accomplished by using network traffic forwarding mechanisms that are provided by the guest OS. For example, in the case of the Linux OS, the *iptables* network utility can be used to introduce a forwarding rule inside the guest OS, to redirect VNF traffic, according to the destination port, to a different network port that is exposed by the VNF-level Mitigation Agent.

The VNF-level Mitigation Agent is transparent to the VNF software. Moreover, the VNF-level Mitigation Agent has only a small impact on network latency and throughput, since it does not perform any traffic analysis or manipulation. The VNF-level Mitigation Agent only computes metrics on network throughput, and sends these metrics to the VNF-level Detection Agent.

When an overload condition occurs, the VNF-level Mitigation Agent filters out part of the input network traffic, in order to protect the VNF software

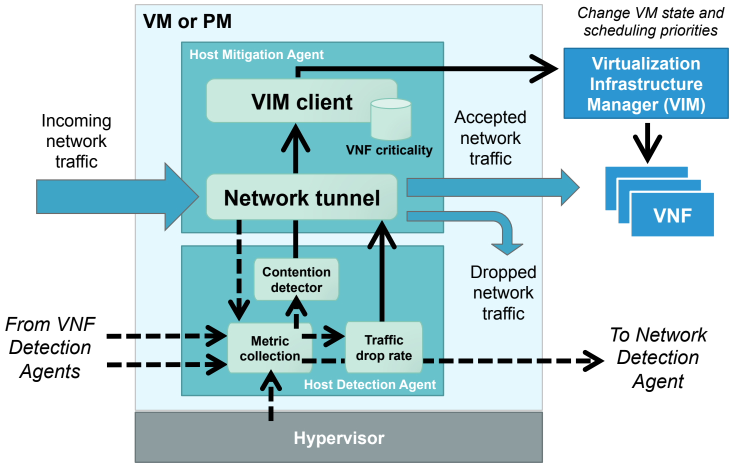


Figure 4.4. Architecture of host-level detection and mitigation.

from the traffic in excess. The traffic in excess is dropped and is not forwarded to the VNF software. The traffic is dropped according to the traffic drop rate configured by the VNF-level Detection Agent.

The VNF-level Mitigation Agent applies a traffic-matching rule on the contents of network traffic (such as, to a "type" field in the header), in order to identify which network traffic it should drop. For example, in the case of the SIP protocol, it is preferable to only drop "REGISTER" and "INVITE" requests in excess, and not to drop other types of messages. In this way, new users are prevented from registering, and the VNF software is protected from the overload caused by new users that try to enter in the network. Moreover, the users that are already registered are not affected by the traffic drop, and do not experience any degradation of the quality of service.

4.2.2 Host-level design

The architecture of the overload control solution for this level is showed in Figure 4.4, which includes a detection agent and a mitigation agent.

Host-level Detection Agent

The Host-level Detection Agent is a multi-threaded application, which can be deployed by the VNF provider in a dedicated VM, in the same cloud infrastructure running the VNFs (Figure 4.2a). An alternative approach, which is viable for the provider of the NFVI, is to run the Host-level Detection Agent on the hypervisor as a privileged process (Figure 4.2b). In both cases, this agent is adopted to detect physical resource contention; in the latter approach, the agent also replaces the VNF-level Detection Agent, in order to protect a VNF from traffic in excess. The Host-level Detection Agent monitors one or more VNFs in the NFV network. It is possible to deploy more than one Host-level Detection Agents on the same cloud infrastructure, where each Host-level Detection Agent monitors a subset of VNFs in the NFV network.

The Host-level Detection Agent receives data on virtual CPU utilization, either from VNF-level Detection Agents (if it is deployed by the VNF provider), through a shared ring buffer or other inter-VM communication channels, or from the hypervisor (if it is deployed by the NFVI provider), using APIs provided by the hypervisor.

The Host-level Detection Agent can detect the traffic in excess towards a VNF, by using the same algorithm of the VNF-level Detection Agent (Alg. 2, and eq. (4.1) and (4.2) in section 4.2.1). It periodically samples the virtual CPU usage of the VM, and the network throughput from the Host-level Mitigation Agent; then, it tunes the traffic drop ratio of individual VNFs to drop traffic. In addition, the Host-level Detection Agent can identify overload conditions that are due to physical resource contention. These conditions may occur when the NFVI experiences a fault (such as, a broken CPU that must be turned off), which reduces the resources available to the VNFs, and which causes competition among them for the remaining resources (but may be insufficient to sustain the current workload). Moreover, physical resource contention can occur due to bad capacity planning and oversubscription of the NFVI [8, 9].

In the case of physical resource contention, it may not suffice to drop traffic, since a VNF would free physical resources that could be consumed by neighbour VMs, causing a vicious circle and worsening the performance of the VNF. In this scenario, the most appropriate course of action is to detect that overload is caused by physical resource contention, and to mitigate the contention by disabling part of the VNFs and by reserving resources for the most critical ones. According to the ETSI NFV resiliency requirements [21, sec. 7.3], NFV is expected to support multiple levels of service availability and, under overload conditions, it should be able to downgrade low priority services and to preempt resources from them (e.g., a video call service should be downgraded or preempted in favor of voice calls).

Under physical CPU contention, a virtual CPU reaches full utilization (i.e., there are no *idle* CPU ticks) even if the workload is below the virtual CPU quota. For example, if two VMs have both a 1 GHz CPU quota, but they both run on an oversubscribed physical CPU (e.g., a 1 GHz physical CPU, with a 2:1 vCPU-to-pCPU ratio), then the hypervisor may be unable to honour the quota, and each VM will actually get up to 0.5 GHz CPU cycles. However, detecting physical CPU contention is problematic for a telecom provider that uses an NFVIaaS, since it has no visibility of the underlying physical host. Moreover, physical CPU contention cannot be detected within the VM using traditional CPU monitoring tools: both in the case of CPU contention and of workload peaks, CPU monitoring tools would report a 100% consumption of the virtual CPU, since they compute the ratio between *busy* and *idle* CPU ticks, and thus would not be able to discriminate between the two cases.

In order to discriminate between physical CPU contention and other overload conditions, and to perform special actions against contention, we consider the absolute number of *busy* CPU ticks (including ticks spent executing both in user-space and kernel-space) that are actually executed by the virtual CPU per unit of time. If there is physical CPU contention, the hypervisor CPU scheduler gives to the virtual CPU less physical CPU cycles than its expected CPU quota. Thus, we detect physical CPU contention by monitoring the number of actual busy CPU ticks of the virtual CPU, and comparing it to the maximum number allowed by its CPU quota:

$$\begin{aligned}\text{pCPU contention} \quad \Leftrightarrow \quad & \text{idle ticks} \approx 0 \\ & \wedge \text{busy ticks} \not\approx \text{maximum busy ticks}\end{aligned}$$

where the maximum for busy ticks is calibrated by running on the virtual CPU a CPU-intensive load under no physical CPU contention. The count of busy ticks can be obtained from the VNF-level Detection Agent inside a VM (section 4.2.1), or from the virtualization infrastructure using hypervisor APIs. The Host-level Detection Agent periodically samples the number of busy ticks since the previous sample, and estimates the physical CPU share allotted to the virtual CPU, by computing the ratio between busy ticks and the amount of "wall-clock" time that has been elapsed. The wall-clock time can be collected by the VNF-level Detection Agent inside a VM (using a paravirtualized clock provided by the hypervisor [91, 92]) and from the virtualization infrastructure. The Host-level Detection Agent notifies the Host-level Mitigation when physical CPU contention arises or disappears.

Finally, the Host-level Detection Agent aggregates the information about the overload state of VNFs that it monitors (either caused by excess traffic, or by physical CPU contention), and sends periodic update messages to the Network-level Detection Agent, as discussed later in this section.

Host-level Mitigation Agent

The Host-level Mitigation Agent is an application that executes in the same environment of the Host-level Detection Agent. It interacts with the Virtualization Infrastructure Manager (VIM) in order to alleviate the contention on physical CPUs, by pre-empting resources from the less important ("non-critical") VMs. The relative importance of VMs is configured according to performance and availability requirements of NFV services (e.g., the ETSI NFV resiliency requirements provide examples of service availability levels, where emergency telecommunications have priority over video streaming and other internet traffic [21, sec. 7]).

The Host-level Mitigation Agent periodically checks the presence of physical CPU contention: if this is the case, it selects the VMs with the lowest criticality, and decreases their scheduling priority in order to free CPU time for the highest-criticality VMs. If the scheduling priority is already at the lowest priority, the VM is suspended. These steps are repeated until the physical CPU contention persists, and reverted when CPU resources are available. This approach can be easily deployed on existing virtualization technologies, such as the KVM hypervisor and OpenStack, using their APIs to change the execution state of VMs.

Optionally, in the case of NFVI providers, such as in NFVIaaS (Figure 4.2b), the Host-level Mitigation Agent can be used to drop the traffic in excess towards individual VNFs, in a similar way to the VNF-level Mitigation Agent (section 4.2.1). This objective is achieved by configuring network traffic forwarding mechanisms of the virtualization infrastructure to establish a network tunnel. When the Host-level Detection Agent detects an overload condition, it can trigger the Host-level Mitigation Agent to drop the traffic in excess. The amount and the type of traffic to drop is configured by the Host-level Detection Agent as described in section 4.2.1: the Host-level Detection Agent updates the traffic drop ratio according to a rule that uses resource utilization metrics, and it applied traffic-matching rules to identify which traffic should be dropped.

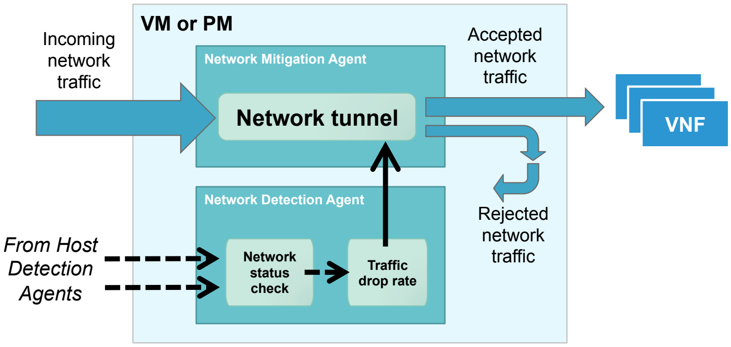


Figure 4.5. Architecture of network-level detection and mitigation.

4.2.3 Network-level design

The architecture of the overload control solution for this level is showed in Figure 4.5, which includes a detection agent and a mitigation agent.

Network-level Detection Agent

The Network-level Detection Agent is a multi-threaded application, which executes in a dedicated VM in the same cloud infrastructure of the VMs running VNF software. Alternatively, it can execute as a privileged process on a physical machine of the NFVI.

The Network-level Detection Agent collects the status of all VNFs in the NFV network, and checks the presence of an overload condition (Alg. 3), according to overload notifications coming from several Host-level Detection Agents. The criteria for detecting a network-level overload condition can be configured by the administrators of the NFV network: a simple criterion is to count the number of VNFs affected by overload, and detect an overload state when overloaded VNFs are the majority. Another possible criterion is to compute a weighted count of the number of overloaded VNFs, by taking into account the relative importance of VNFs in the NFV network.

In a similar way to the VNF-level, the Network-level Detection and Mitigation Agents protect the NFV network from the input network traffic in excess. Since these agents are deployed at the boundary of the VNF network, they can prevent new users from entering the VNF network by explicitly rejecting them, which differs from traffic drops inside individual VNFs or hosts, where traffic is dropped without notifying the user. The traffic is rejected according to a *traffic rejection rate*, which is controlled by the Network-level Detection Agent.

The traffic rejection rate is gradually increased when the VNFs are in an overload condition, and it is decreased otherwise. It is periodically updated according to a configurable, multiplicative function:

$$\text{capacity} = \begin{cases} \text{capacity}/(\alpha+\gamma) & \text{if overloaded} \\ \text{capacity}\cdot(\beta-\gamma) & \text{otherwise} \end{cases} \tag{4.3}$$

$$\text{reject\_rate} = 100\cdot\left(1-\frac{\text{capacity}}{\text{incoming\_traffic}[N]}\right)[\%] \tag{4.4}$$

in which the *reject_rate* is capped between 0% and 100%. The *incoming_traffic* is the volume of traffic in input, and α and β are constants, with $\alpha > 1$ and $\beta > 2$. The γ coefficient is a variable factor, which tunes the reject rate according to the persistence of the overload condition. It is defined as:

$$\gamma = \frac{\text{dropped\_traffic}[N]}{\text{incoming\_traffic}[N]} \tag{4.5}$$

that is, γ represents the fraction of traffic that has been rejected during the last sampling period. This coefficient has been introduced to keep the reject rate low if the overload condition lasts for a short amount of time, in order to soften the impact of sporadic false positives in overload detection; and, at the same time, this coefficient serves to keep the reject rate high if the overload condition is severe and persists over time. When the current fraction γ of rejected traffic is null or low, $\alpha+\gamma$ is closer to 1, thus the capacity decreases with a smaller step, and increases with a larger step. Thus, the approach avoids to reject too much traffic when the overload condition is short and sporadic. Instead, when the γ is high (which happens when the overload condition already lasted for a relatively long time), the capacity decreases with a larger step, and increases with a smaller step. In this way, if the overload condition disappears only for a small amount of time, the reject rate is still kept high; the full capacity is restored only once the network becomes stable and non-overloaded.

Algorithm 3: Network-level detection and mitigation

```
Data: $SP$: sampling period
Data: $VNF[1 \dots M]$: overload state of monitored VNFs
Result: reject_rate for incoming VNF Network traffic
begin
    while True do
        forall monitored VNFs do
            collect overload state for $VNF[k]$
        if majority of VNFs is overloaded then
            decrease capacity;
        else
            increase capacity;
        update and send the reject_rate to the Network-level
          Mitigation Agent;
        wait $SP$ seconds;
```

Network-level Mitigation Agent

The Network-level Mitigation Agent acts as a network tunnel at the boundary of the NFV network. The Network-level Mitigation Agent receives the traffic that was originally intended for the NFV network, and forwards it to the

VNFs.

This forwarding is accomplished by installing the Network-level Mitigation Agent into a load balancer, placed at the boundaries of the NFV network, either in a dedicated VM, or on a physical machine. Therefore, the Network-level Mitigation Agent is transparent to the VNFs. Moreover, the Network-level Mitigation Agent has only a small impact on network latency and throughput, since it does not perform any traffic analysis or manipulation. The traffic in excess is dropped and not forwarded to VNFs. Moreover, the Network-level Mitigation Agent can reject input traffic by replying with an overload notification to clients, in order to prevent them to generate more network traffic. For example, in the case of the SIP protocol, the Network-level Mitigation Agent can reply with a "*503 Service Unavailable*" response in order to notify clients about the overload state. Moreover, the Network-level Mitigation Agent applies a traffic-matching rule on the contents of network traffic (such as, to a "type" field in a packet header), in order to identify which network traffic it should drop (such as, session initiation requests).

4.3 Experimental evaluation on an NFV IMS

To evaluate the overload control framework, we performed an experimental analysis on the NFV-oriented IP Multimedia Subsystem (IMS), Clearwater. This analysis is aimed at evaluating the ability of the overload control framework in the context of a real NFV software, in terms of performance under overload conditions, overhead of the framework, and failures of NFV software.

4.3.1 Testbed and technical implementation

The experimental testbed consists of four host machines: three Dell PowerEdge R520 servers, equipped with two 8-Core 2.2 GHz Intel Xeon CPU, 64GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber

Channel HBA; one Dell PowerEdge R320 server equipped with a 4-Core 2.8 GHz Intel Xeon CPU, 8GB DDR3 RAM, two 500GB SATA HDD, two 1-Gbps Ethernet NICs, 8-Gbps Fiber Channel HBA; A PowerVault MD3620F disk array with 4TB of network storage with a 8-Gbps Fiber Channel link.

The hosts are connected to a 1-Gbps Ethernet network for general-purpose traffic, and another 1-Gbps Ethernet network for management traffic. The virtual disks of VMs are stored on three distinct GlusterFS partitions of the PowerVault SAN, which are mounted on the hosts through the Fiber Channel link.

The hosts are configured with CentOS Linux 7 and the KVM hypervisor. The testbed is managed using the *OpenStack* virtualization platform, version Juno [90]. The Dell PowerEdge R320 serves as OpenStack Controller and Network node; the three Dell PowerEdge R520 servers represent the OpenStack Compute and Storage nodes, and run the VMs of the Clearwater IMS. The OpenStack services include: *Nova*, which manages the compute domain; *Neutron*, which manages virtual networks among VMs; *Cinder*, which controls the lifecycle of VM volumes; *Glance*, which manages the cloud images of VMs; *Heat*, which orchestrates, through a native REST API, the virtual IMS deployment; *Horizon*, which supports the Web-based management dashboard.

The agents of the *NFV-Throttle* framework are deployed both on the VMs (VNF-level agents) and on the hosts (Host-level and Network-level agents). The agents have been developed in C, respectively as background daemons at VNF-level and Host-level, and as extensions of the OpenSIPS [93] proxy at Network-level. The VNF-level and Host-level detection agents collect CPU utilization metrics from the *proc* FS, and network utilization metrics from the mitigation agents. The mitigation agents, both at VNF-level and at Host-level, act as a filtering proxy for all the network traffic destined to a specific VNF, using *iptables* NAT rules. The network traffic accepted by the agent is forwarded to the VNF, on behalf of the originator. In the case of UDP traffic (such as in the case of SIP clients and P-CSCF VNF instances), UDP datagrams are forwarded by the agent on behalf of the source, by replacing the source IP and port with the ones of the SIP client; thus, the destination (e.g., P-CSCF) can directly reply to the source. In the case of TCP traffic (such as between a P-CSCF VNF and a S-CSCF VNF), during the TCP handshake phase, the mitigation agents establishes two connections, respectively with the source (e.g., P-CSCF) and the destination (e.g., S-CSCF). Then, the agent reads and writes data from both the two streams, acting as a man-in-the-middle. Note that, for both UDP and TCP, only a fixed set of *iptables* rules is required, regardless of the number of clients and connections.

4.3.2 Experimental plan

We evaluate the proposed overload control framework in the context of the Clearwater IMS case study, by performing experiments with stressful workloads and resource contention. In particular, we evaluate:

- The ability of the framework to assure a high throughput (up to the maximum capacity of the system), in terms of **RAPS** (register attempts per second) and **CAPS** (call attempts per second) that are successfully handled with no failures (i.e., requests that are neither timed-out nor rejected).
- The resource overhead introduced by the framework, in terms of **CPU** and **memory footprint** consumed by the agents of the overload control framework.

The experiments use a mix of SIP registrations and call setup requests. The workload is generated using the *SIPp* traffic generator [94] to emulate SIP subscribers. Each couple of subscribers will attempt to register or renew the registration every 5 minutes on average. After a successful registration, a subscriber can either attempt to setup a call to the other (with 16% of probability), or remain idle until the next registration renewal (with 84% of probability). The call hold time is configured to 60s. We calibrate the number of VNF instances with a preliminary *capacity planning* using 400k subscribers. These numbers have been suggested by our industrial partners as a realistic baseline for testing an IMS service, and on which we impose overload conditions.

We tuned the number of VNF instances to have at most 80% virtual CPU utilization (which is measured by sampling the average CPU utilization every minute), and no failures. The IMS can handle this workload with 10 replicas of Bono, Sprout, and Homestead, 4 replicas of Ralf, and 1 replica of Homer and DNS. Each replica runs on a distinct VM with 1 virtual CPU. In this experimental setup, 400k subscribers represent the **engineered capacity** of the IMS (see also Figure 4.1). The IMS experiences an overload condition when the number of subscribers exceeds this engineered capacity ($\gg$ 400k subscribers). In these cases, the CPU becomes the performance bottleneck. Moreover, an overload condition happens when the IMS compete for resources with other services that are deployed on the same physical infrastructure.

We consider three high-workload scenarios to evaluate the performance of overload control under a peak of subscribers. Every scenario is executed four times, respectively with: the plain Clearwater IMS; the VNF-level overload control; the Host-level overload control; and the Network-level overload control. In total, we perform 12 high-workload experiments. We adopt the following workloads (Table 4.1):

- **Small overload** (480K subscribers): the load is at 120% with respect to the engineered level, and saturates the maximum capacity of the testbed. At this load level, the overload control solution should throttle a small part of service requests, in order to preserve the QoS for subscribers that are already registered in the IMS before the overload.

- **Medium overload** (1M subscribers): the load is 250% with respect to the engineered level, and above the maximum capacity of the testbed.

At this load level, the overload control solution should throttle a large amount of requests to prevent a significant throughput degradation.

- **High overload** (4M subscribers): the load is ten times higher (1000%) than the engineered level. At this load level, there is a significant resource pressure since a considerable amount of connections must be handled, thus exposing the IMS software to potential crashes due to resource exhaustion.

Each experiment lasts 1 hour, and is divided in three phases:

- **Load generation (Ramp-up)**: When the experiment starts, 400k subscribers are created in the initial 15 minutes (*Initial ramp-up period*). The system can handle this load without failures. This load is generated by a set of 10 SIPp instances, for all the duration of the experiment. This phase is common to all the experiments.
- **Overload generation**: This phase starts at the 20th minute, and lasts for 30 minutes. In this phase, additional subscribers, over the engineered level (Table 4.1), are introduced in a short amount of time (*Overload Ramp-up period*). Then, all the subscribers constantly generate requests for call setup and registration renewal.
- **Overload termination (Ramp down)**: This phase starts at the 50th minute, and lasts until the end of the run. In this phase, each subscriber that fails to register or make a call, will not attempt to retry and will leave the system.

Moreover, we consider an additional experimental scenario, in which the overload condition is caused by resource contention between the IMS and other services deployed on the same physical infrastructure. In this scenario, the Host-level agents have to shield the IMS from resource contention, by throttling other services. Thus, we perform this additional experiment:

Table 4.1. Workloads used to evaluate the overload control solution.

# Subscribers	Load Level	RAPS	CAPS
400k **(Engineered Level)**	**100%**	**1,379**	**111**
480k (Small Overload)	120%	1,655	133
1M (Medium Overload)	250%	3,448	278
4M (High Overload)	1000%	13,793	1,111

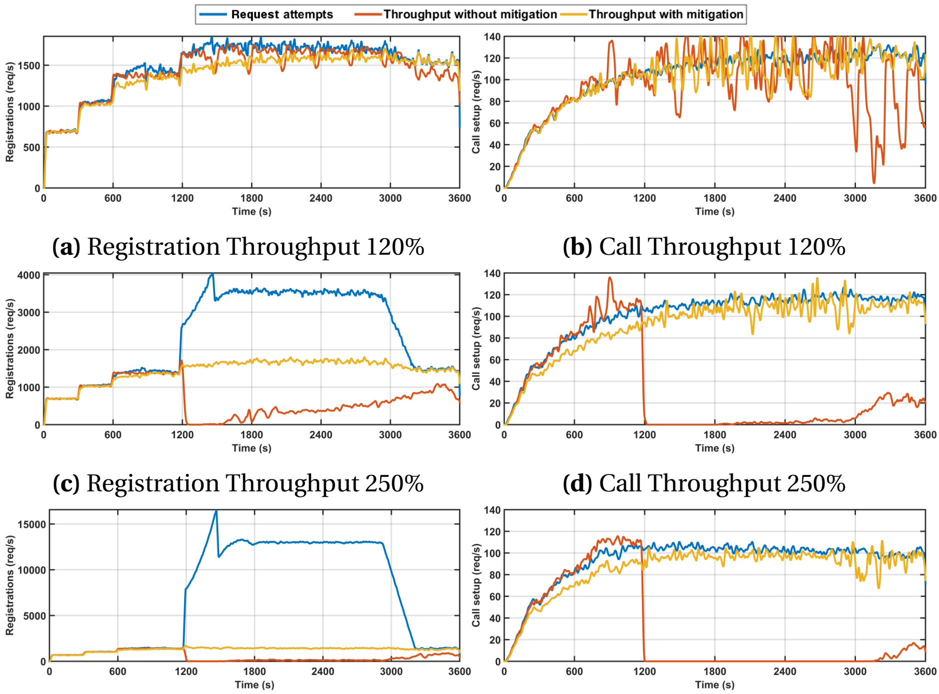
- **Resource contention**: At the steady state, the load level is 350k subscribers, close to the engineered level. After 20 minutes, we run a "CPU hog" on one of the three physical hosts (using the *cpuburn* tool [95]), causing CPU contention between the processes and VMs running on that physical host.

4.3.3 Experimental results

Node level

We first consider the case in which overload control is performed only at the VNF-level (i.e., by installing an agent inside VMs, as discussed in Section 4.2.1). Figure 4.6 shows the performance of the Clearwater IMS at varying levels of overload, respectively at 120%, 250% and 1000% load with respect to the engineered capacity.

The graphs of Figure 4.6 show the registration throughput on the left side, and the call throughput on the right side. Each graph shows three curves: the input load, in terms of registration and call requests per second, and the throughput of successful requests, respectively without and with overload control at VNF-level.



(a) Registration Throughput 120%

(b) Call Throughput 120%

(c) Registration Throughput 250%

(d) Call Throughput 250%

(e) Registration Throughput 1000%

(f) Call Throughput 1000%

Figure 4.6. Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Node Level.

With an overload level of 120% (Figure 4.6a and 4.6b) the registration and call throughput are close to the input request rate, both with and without the proposed overload control framework. In both cases, the capacity of the IMS has been saturated. However, in the case without our overload control framework, the call throughput exhibits a significant variability, and tends to be lower than the input rate of requests. This behavior is a consequence of the problem discussed in Section 4.1: even if resources are fully utilized, they do not necessarily produce useful work, since the system attempts to manage too many users but cannot provide an acceptable QoS to any of them. Instead, the overload control solution has been able to avoid service failures for already-established sessions, by rejecting the requests in excess during the overload phase.

With higher overload levels (Figure 4.6c – 4.6d at 150% load, and Figure 4.6e – 4.6f at 1000% load), and without our overload control framework, the impact of overload is even more severe. We observed that most of the nodes exhibit failures due to resource exhaustion, causing the crash of VNFs and performance degradation of the IMS. This results in a significant performance degradation, with registration throughput lower than 200 RAPS and call throughput lower than 1 CAPS in the worst case.

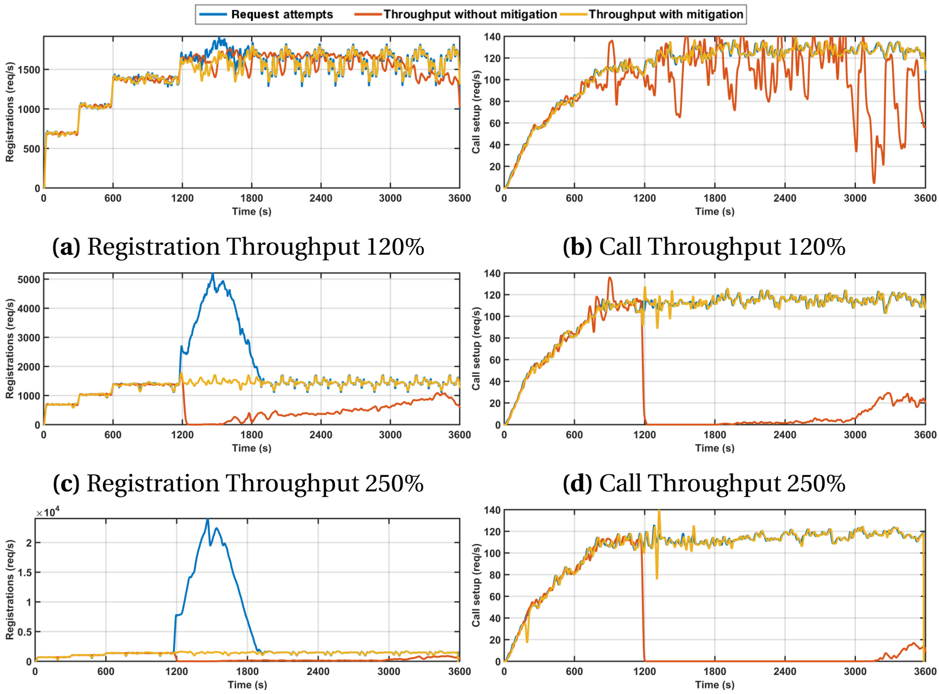
In the same scenarios, with the overload control framework, the registration and call throughput are stable around the engineered level, which is the maximal throughput attainable by the IMS. Moreover, there are no failures of the VNFs, since the overload control framework is implemented outside VNF software and is more robust to huge overload conditions. In particular:

- Load level of 250%: (1) the registration throughput reaches on average **1664.9 RAPS**, which is **137%** more than the case without the mitigation and **20%** more than the engineered level; (2) the call throughput reaches on average **114.60 CAPS**, which is **194%** more than the case without the mitigation and close to the engineered level.
- Load level of 1000%: (1) the registration throughput is **1439.5 RAPS**, which is **152%** more than the case without the mitigation and **4%** higher than the engineered level; the call throughput reaches on average **97.17 CAPS**, which is **200%** more than the case without the mitigation and close to the engineered level.

Host level

We performed the same experiments using the Host-level overload detection and mitigation, which replace their VNF-level counterparts. The results obtained with Host-Level overload control are comparable to VNF-level overload control in Figure 4.6, thus they are not showed here for the sake of brevity. In summary, the overload control framework, under an overload level of 250%, can achieve a registration throughput **136%** higher than the case without the mitigation, and **18%** more than the engineered level. Similar results were also obtained under an overload level of 1000%. Again, overload control prevented IMS failures due to resource exhaustion.

Network Level



(a) Registration Throughput 120%

(b) Call Throughput 120%

(c) Registration Throughput 250%

(d) Call Throughput 250%

(e) Registration Throughput 1000%

(f) Call Throughput 1000%

Figure 4.7. Registration and Call Throughput for each overload level (i.e., 120%, 250% and 1000%) at Network Level.

Figure 4.7 show the performance measurements obtained with Network-level overload control. As for the previous cases (VNF-level and Host-level), the overload control at Network-level is able to sustain a high throughput in all overload scenarios. Moreover, with a load level of 250% and 1000%, the control solution is able to avoid resource exhaustion and crashes of IMS components.

These experiments point out an additional benefit of Network-level overload control. During the overload, the Network-level mitigation agent rejects the registration requests in excess by replying to the clients. During the first 10 minutes of overload (in the period 1200s-1800s in the graphs), the rate of incoming registration requests gradually decreases due to rejections, and stabilizes again around the engineered capacity.

Figure 4.8 summarizes the results, by providing aggregated statistics (median, upper and lower quartiles, minimum and maximum) obtained respectively with overload control at VNF-level, Host-level, and Network-level. Figure 4.8a shows the performance of the IMS in terms of registration throughput at different loads (from 400k to 4M subscribers), while Figure 4.8b shows the performance of the IMS in terms of call throughput.

At the engineered level (400k subscribers), there are no significant differences between the three cases and the engineered capacity (Table 4.1). In all cases, during the first 20 minutes of the experiments, when the load is within the engineered level, the performance with and without overload control are closely matching. Thus, the overload control does not have negative side effects on the IMS when there is no overload condition.

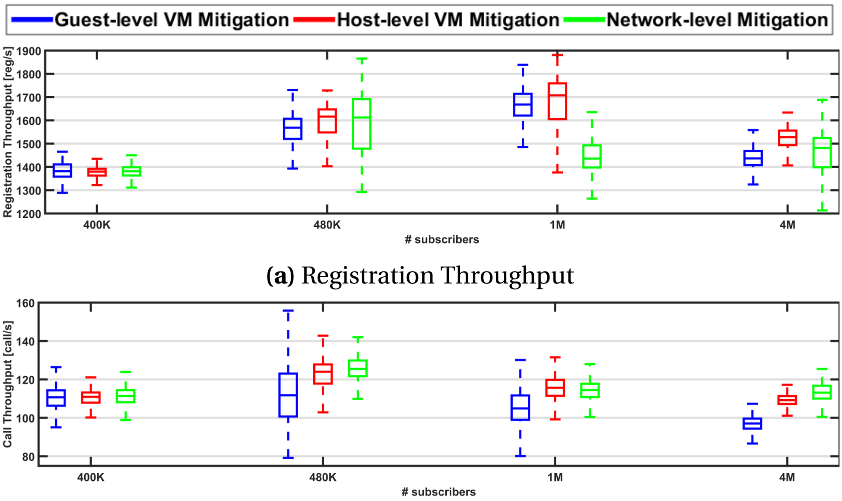
In overload conditions, the Host-level overload control provides the best average registration and call throughput compared to VNF-level and Network-level control. The performance gap between VNF-level and Host-level can be explained by observing that the VNF-level control incurs in the overhead of transmitting all of the traffic to VMs, and to discard the traffic in excess in the VM. The Host-level solution acts in the hypervisor rather than the VM, thus avoiding this additional overhead. Thus, when feasible, the Host-level solution should be preferred to the VNF-level one. The Host-level solution can be adopted in the case of NFVIaaS *providers*, which have access to the infrastructure, while it may not be feasible for NFVIaaS *consumers*, which can only deploy VNF-level solutions.

The Network-level overload control also exhibits lower performance than the Host-level one, in particular with respect to the registration throughput. The performance of Network-level overload control is mainly affected by the detection mechanism. The main factor is that detection is distributed and uses a longer sampling period compared to the Host-level solution (which are respectively $30s$ and $10s$), since the Network-level solution needs to collect information from several nodes. Thus, the Network-level solution has a slightly higher detection latency, and it is thus more exposed to oscillations of the workload. Moreover, there are sporadic cases in which the workload is not uniformly balanced across the replicas. Since the Network-level solution detects an overload when a majority of nodes is overloaded, these cases lead to sporadic delays in overload detection.

Resource contention

To complete the evaluation of the Host-level overload control solution, we performed an experiment in which the overload is caused by an additional workload that shares the physical infrastructure with the IMS.

Figure 4.9 shows the registration attempt rate (which is approximately constant for most of the experiment), and the registration and call throughput, both without mitigation (red lines) and with the Host-level mitigation (yellow lines). Without mitigation, the overload condition (starting after 1200s) halves the registration throughput, and significantly reduces the call throughput. With Host-level mitigation, the CPU contention is relieved by reducing the priority of the “CPU hog”, until the throughput of the IMS stabilizes again



(a) Registration Throughput

(b) Call Throughput

Figure 4.8. Mitigation performance at different operational levels (i.e., node, host and network level)

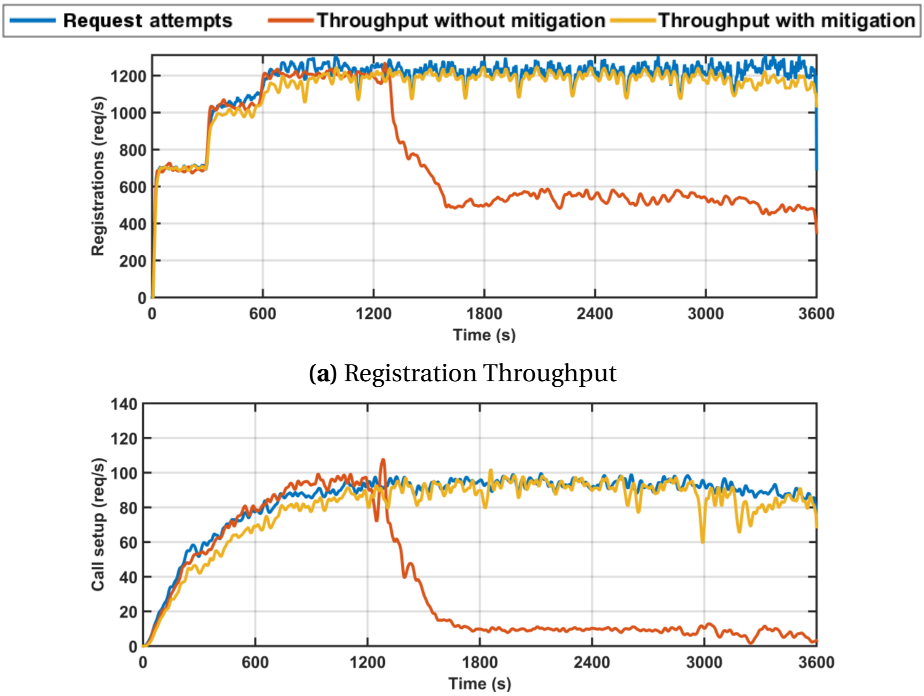
around the engineered capacity.

Overhead Evaluation

The mitigation agents act as a lightweight filtering proxy for the network traffic destined to VNFs. Since all the traffic, both TCP and UDP, will be processed by the mitigation agent, we analyze the performance overhead of this critical component. Moreover, since the implementation of the proxy agent is different between the UDP and the TCP transport protocols, we separately analyze both of them.

We performed experiments with the High Overload scenario (that is, 1000% the engineered level), as discussed in Section 4.3.2. This scenario is the most stressful among our experiments, and thus represents a worst-case for our overload control framework.

The plot in Figure 4.10 shows the CPU consumption of the mitigation



(a) Registration Throughput

(b) Call Setup Throughput

Figure 4.9. Overload control results at Host level for pCPU contention.

agent during the experiments when processing UDP datagrams. When the workload is within the engineered level of traffic (900s-1200s), the CPU consumption is very little (≈ 1.5%). When the workload reaches ten times the engineered level, the mitigation agents drops the UDP traffic in excess, and its overhead is less than 4%. The memory consumption during the whole experiment is fixed at 3.5MB, since the agent does not allocate any dynamic memory.

We repeated the same experiment, at maximum level of overload (i.e., 1000% the engineered level), by analyzing the overhead under TCP traffic. The plot in Figure 4.11 shows the CPU consumption of the mitigation agent during

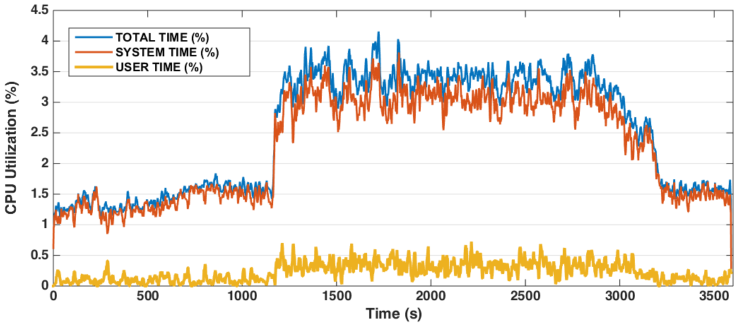


Figure 4.10. CPU Consumption of the UDP mitigation proxy

the experiment. At the engineered level of traffic (900s-1200s), the overhead is again very little (≈ 1.5%). At ten times the engineered level, when dropping the TCP traffic in excess, the overhead of the mitigation agent is less than 3%. In the case of TCP, the memory consumption is dependant of the number of TCP connections that are currently active. Even during the peak of the traffic, the maximum memory consumption was only 16MB.

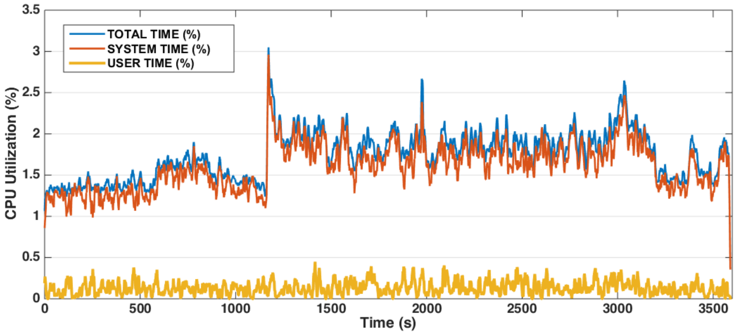


Figure 4.11. CPU Consumption of the TCP mitigation proxy

4.4 Managing the threats of the physical CPU contention at guest-level

This section analyzes the problem of overloads caused by physical CPU contention in cloud infrastructures, from the perspective of time-critical applications (such as Virtual Network Functions) running at guest level. In this particular scenario, overload control solutions to counteract traffic spikes (e.g., traffic throttling) are counterproductive against overloads caused by CPU contention. Then I propose a general guest-level solution to protect applications from overloads also in the case of CPU contention. We reproduced the phenomena on the Clearwater IMS testbed. The results show that the approach can dynamically adapt the service throughput to the actual system capacity in both cases of traffic spikes and CPU contention, by guaranteeing at the same time the IMS latency requirements.

4.4.1 Overview of CPU overloads and CPU utilization metrics

In this section, we expose the problem of overload conditions, how to interpret CPU utilization metrics, and the pitfalls for overload control strategies when using these metrics.

Ideally, the input traffic for a service should not exceed its **engineered capacity**, that is, the maximum amount of input traffic that can be served while achieving SLAs. SLAs typically require a low probability of failures (such as, traffic loss or processing errors) and low latency (such as, the time to process or respond to an individual traffic unit). These requirements are especially demanding in the case of the telecom domain [22, 96], where the engineered capacity is carefully planned at design time, by allocating computing resources according both to cost considerations, and to the expected **reference workload**: for example, according to the expected rate of *busy-hour call attempts* (BHCA) in the case of a VoIP service.

In the context of IaaS, the designers of VNFs need to plan in advance the flavor and the expected amount of VMs; for example, a common rule-of-thumb is to plan for VMs such that each VM consumes at most 90%, or some other threshold (the **engineered level**), of the available virtual CPUs under the reference workload, leaving a small amount of residual capacity as a factor of safety [3, 97]. Overload conditions saturate the capacity of virtual CPUs; in these cases, the VNFs should **throttle** the input traffic (i.e., rate-limit by dropping or rejecting requests) in order to assure that the traffic processed by the VNFs is within the engineered capacity and can meet the SLAs. This strategy is further discussed in Section 4.4.2.

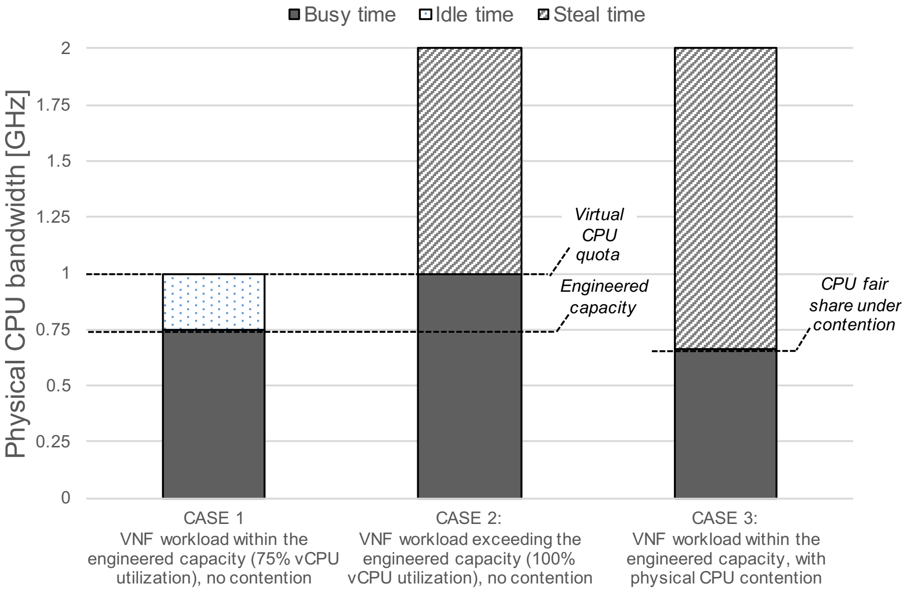


Figure 4.12. CPU utilization metrics under three scenarios.

Physical resource contention is a special case of overload condition, in which the available capacity of the VNFs is reduced due to competition. However, the behavior of the system is different than the case of traffic spikes. To illustrate the problem, we consider thorough the paper a generic example (in Figure 4.12) of a VNF with a 1-GHz virtual CPU, deployed on a 2-GHz physical CPU. Therefore, the CPU quota of the VM is 50% of the physical CPU. In this example, we assume that the engineered capacity of the VNF uses 75% of the virtual CPU under the reference workload. From inside the VM (Figure 4.12, CASE 1), the OS measures the virtual CPU utilization by counting the virtual CPU cycles that have been spent *busy* at executing applications or the OS kernel, and *idle* at waiting for I/O or without any workload (i.e., **vCPU utilization** = $^{\text{busy}}/_{\text{busy+idle}}$). When the input traffic overloads the VNF (Figure 4.12, CASE 2), the virtual CPU utilization raises to 100% to serve all of the traffic, and hits the CPU quota at 1 GHz enforced by the hypervisor.

In addition to these metrics, we also consider the **CPU steal time** metric, which is also influenced by overload conditions, but can be mistakenly considered as an indicator of physical CPU contention. We use the name "CPU steal" in reference to the metric available in Linux and in the KVM and IBM z/VM hypervisors [98, 99, 100]; an equivalent metric is also available in other hypervisors such as VMware ESXi, Xen and Microsoft Hyper-V, respectively under the name "CPU Stolen Time" [101, 102] and "CPU Wait Time Per Dispatch" [103]. This metric is provided by hypervisors to VMs, e.g., through hypervisor calls. In all these systems, this metric is technically defined as *the time that a virtual CPU is ready to execute, but it is waiting to execute on the physical CPU.* In other terms, the metric represents the time spent by the virtual CPU on the hypervisor's scheduling queue. The term "steal" refers to CPU cycles that a VM spends waiting because either the hypervisor or other VMs are using the physical CPU (for example, the hypervisor is using CPU cycles to emulate an I/O device). However, in most situations no CPU cycle is actually "stolen" from the VM, as the hypervisor still assures the CPU quota for the VM, and that the VM is eventually scheduled; it would be better understood as an "involuntary wait" time. For example, in the CASE 2 of Figure 4.12, the

VM is put on hold after that it consumes its virtual CPU quota; thus, the rest of the physical CPU time is accounted as "steal time" from the perspective of the VM, since it is waiting on the scheduling queue. Even in the case of low workload, it is still possible that a moderate share of CPU time is accounted as stolen, e.g., when two VMs are sporadically ready to execute at the same time. Thus, steal time is not a sufficient condition for an overload condition.

The third scenario involves physical CPU contention (Figure 4.12, CASE 3). In this case, we are assuming that 3 VMs with equal priority are scheduled on the same physical CPU (e.g., because of overcommitment, bug or misconfiguration of the infrastructure). The 3 VMs all have a CPU quota set to 1 GHz, and an engineered capacity that uses 75% of the virtual CPU (as in the previous two scenarios). Since the total CPU demand ($0.75 \cdot 3$ GHz) exceeds the capacity of the physical CPU (2 GHz), the hypervisor equally divides the CPU bandwidth among the VMs, where each virtual CPU actually gets a slice (**fair share**) of 0.66 GHz (i.e., 33% of the physical CPU time). Since the VNF is ready to execute even after consuming this slice (as the workload exceeds the virtual CPU capacity), the rest of the physical CPU time (66%) is accounted as steal time for the VM.

Both in CASE 2 and CASE 3 of Figure 4.12, the VNF is in an overload condition. However, if the VNF is deployed on IaaS, it cannot easily distinguish between the two cases, since the VNF cannot inspect or control the underlying infrastructure. From the perspective of the VNF, only looking for high virtual CPU consumption or for high CPU steal time does not suffice to discriminate between a traffic spike or physical resource contention. The only difference between the two cases is that the actual CPU share of the VM (0.66 GHz) is lower than the original CPU quota (1 GHz). Therefore, to address both these cases, the proposed overload control approach throttles the workload by adapting to the CPU share (either the quota or the fair share) that is actually available to the VNF.

4.4.2 Mitigation strategy

To recover from overload conditions, the long term solution would be to meet the high demand by scaling up the computing resources, or to relieve physical resource contention by shutting down other services that have a lower priority or that are hogging the resources. However, these recovery actions can take several minutes, even in an optimistic case. During this transient period, VNFs are still exposed to the risk of outages. If the VNFs attempts to serve much more traffic than their capacity, then each traffic unit will not be served with enough computing resources to meet SLA requirements. As a result, the *useful* throughput of the VNFs (i.e., the rate of successfully processed traffic) can significantly degrade [104, 45]. Moreover, handling too much traffic at the same time increases the likelihood of *VNF software failures* such as failed resource allocations, timeouts, and race conditions [105, 106].

Therefore, long-term recovery actions should combined with short-term solutions for throttling the traffic, in order to let in the system only the traffic that can be processed with the currently available capacity [30, 107, 31, 44]. In the case of traffic spikes in VNFs (the CASE 2 in Figure 4.12), the throttling algorithm should reject part of the traffic, in order to reduce the virtual CPU utilization to the engineered capacity (i.e., to return to the CASE 1 in Figure 4.12). For example, *increase/decrease algorithms* are a popular solution to tune the amount of traffic to be accepted (e.g., the window size for packet flow control) [108, 109, 110] by decreasing the traffic when the network is overloaded (e.g., by a constant or multiplicative factor), or by increasing the traffic otherwise. This approach has been recently applied in the context of NFV [111], using a heuristic criterion to tune the traffic that a VNF can serve (*capacity*):

$$\text{capacity} = \frac{\text{processed\_traffic}}{\text{current\_vcpu\_usage}} \cdot \text{reference\_vcpu\_usage} \tag{4.6}$$

where the first factor estimates the cost per traffic unit (in terms of virtual CPU cycles), which is multiplied by the reference virtual CPU budget (i.e., the engineered level) to get the total amount of traffic that can be correctly served. When the virtual CPU utilization exceeds the engineered level, the heuristic drops a percentage of the incoming traffic (*drop rate*):

$$\text{drop\_rate} = 100 \cdot \left(1 - \frac{\text{capacity}}{\text{incoming\_traffic}}\right) \tag{4.7}$$

in which the higher the gap between capacity and the incoming traffic, the higher the drop rate. The drop rate is periodically updated every few seconds, and is capped between 0% and 100%. In the case of a traffic spike, the virtual CPU utilization increases, thus the heuristic lowers the *capacity* and increases the *drop rate*; as result, the virtual CPU utilization settles again around the engineered level.

Nevertheless, this heuristic may not work correctly in the case of physical CPU contention. We consider again the example of Section 4.4.1, where the physical CPU contention leads to the following chain of events (see also the Figure 4.13):

1. Due to the contention, the hypervisor allocates less physical CPU time to the VM (0.66 GHz, as in the CASE 3 in Figure 4.12). As a result, the current workload saturates the VNF, and the virtual CPU utilization becomes 100% (i.e., the ratio $^{\text{busy}}/_{\text{busy+idle}}$), which is higher than the reference CPU utilization (e.g., 75% in the example).

2. The heuristic increases the drop rate to reduce the load. The virtual CPU utilization then settles around 75%. It is important to note that the 75% of the virtual CPU is equal to $0.66 \cdot 75\% = 0.5$ GHz of physical CPU. The residual 25% of the virtual CPU (i.e., $0.66 \cdot 25\% = 0.166$ GHz of physical CPU) becomes idle.

3. Due to the physical CPU contention, the hypervisor opportunistically

schedules these idle CPU cycles for the demand of other VMs or processes on the host machine. Thus, the virtual CPU is not anymore idle, and virtual CPU utilization becomes again 100%.

4. The heuristic further increases the drop rate, to reduce again the virtual CPU utilization down to 75% (as in the previous step 2). The virtual CPU now consumes $0.66 \cdot 75\% \cdot 75\% = 0.375$ GHz of physical CPU.

5. The hypervisor preempts again the idle CPU time. The heuristic enters a vicious cycle where the virtual CPU utilization is reduced more and more.

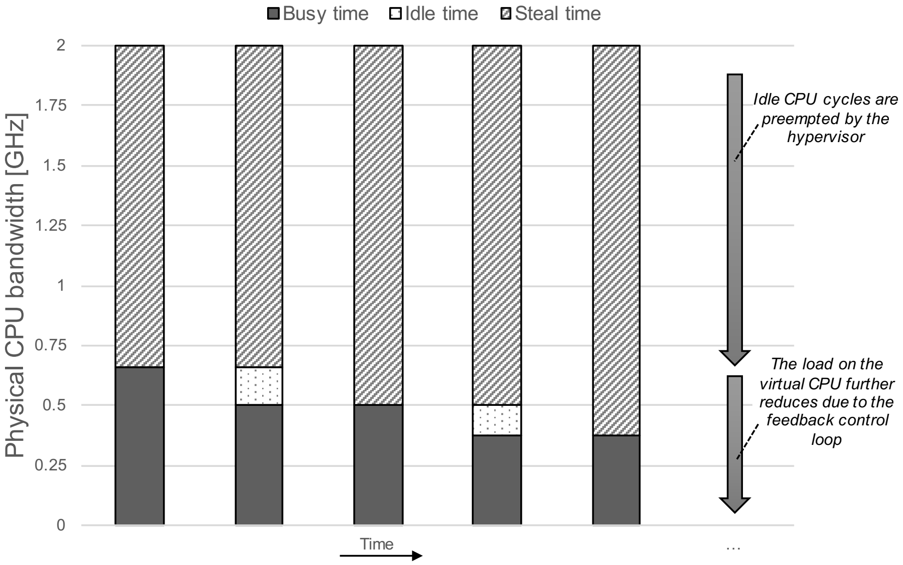


Figure 4.13. Chain of events caused by physical CPU contention.

The vicious cycle is caused by the *work-conserving* behavior of hypervisor schedulers (i.e., they ensure that the CPU is never idle if there is at least one VM ready for execution) [112, 113, 114]. The VNF yields to the hypervisor part of its virtual CPU time, by dropping part of the incoming traffic. In the case of physical CPU contention, in which several VMs or processes on the host machine are demanding more CPU time than the available physical CPU, the hypervisor scheduler uses the freed CPU cycles to meet these demands. Then, the virtual CPU shrinks again and causes the vicious cycle. In general, the feedback control loop approach (not limited to the heuristics of eqs. (4.6) and (4.7), but any other control rule based on virtual CPU utilization) can be vulnerable to physical CPU contention, due to the distortion of virtual CPU utilization metrics.

To address the problem of overload control under physical CPU contention, we extend the feedback control loop approach with an additional mechanism to break the vicious cycle. The design goal of the approach is to assure that the VNF gets no less than its fair share of the physical CPU even under contention (e.g., 0.66 GHz in the previous example); and, at the same time, that the virtual CPU utilization inside the VNF settles at the engineered level (e.g., 75% of the virtual CPU in the example). This condition is showed in Figure 4.14: the available virtual CPU under physical contention reduces to 0.66 GHz; since this virtual CPU is not sufficient to reach the original engineered level (0.75 GHz), we still apply the feedback control loop to reduce the virtual CPU utilization down to 75% of the available virtual CPU (i.e., 0.5 GHz of physical CPU). This is the same condition of the step 3 of the vicious cycle; we break the cycle at this point, using the following approach.

We introduce a mechanism into the VNF to avoid the preemption of idle virtual CPU cycles under physical CPU contention. This effect can be obtained in different ways depending on the guest OS used in the VNF. The most generic approach is to add a *placeholder* process (one process per virtual CPU of the VM) that actively consumes virtual CPU cycles to avoid preemption by the hypervisor; it executes a CPU-bound task for the sake of consuming virtual CPU cycles. The placeholder process should execute at minimal priority on

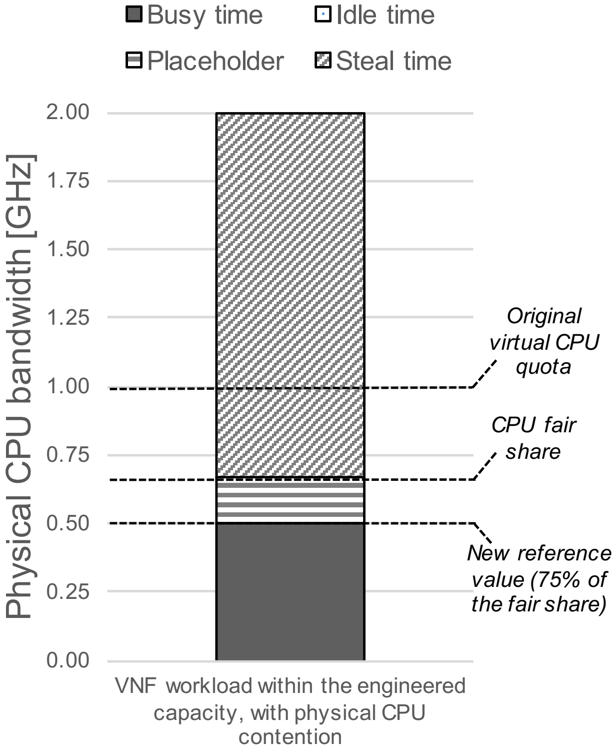


Figure 4.14. CPU utilization metrics under physical contention, with virtual CPU placeholder.

the guest OS of the VNF; moreover, it should be configured as a *batch* task in order not to take away any virtual CPU cycle from the VNF software (i.e., the placeholder only uses the virtual CPU when the VNF is not executing). For example, this effect can be obtained on Linux by setting the `SCHED_BATCH` or `SCHED_IDLE` scheduler class for the task [115], and on Windows by setting an idle trigger [116]. Yet another approach is to configure or to modify the idle loop of the guest OS [117]. As a result, the placeholder takes the place of the idle time of the virtual CPU, as in Figure 4.14: at any given time, the virtual

CPU is either executing the VNF or the placeholder process, and the virtual CPU consumes the residual physical CPU cycles granted by the hypervisor scheduler. This behavior breaks the vicious cycle, since the hypervisor cannot preempt the virtual CPU cycles that are freed by the feedback control loop. Moreover, settling the virtual CPU utilization at the engineered level provides a “margin of safety" (e.g., to compensate for small random workload fluctuations) as in the case of the original engineered level, since the VNF software can preempt the placeholder process at anytime.

We enable the placeholder process on the condition that the CPU steal time spans all the physical CPU not used by the VM. This condition occurs when the VNF consumes its available virtual CPU, either because of a traffic spike that saturates the virtual CPU quota (CASE 2 in Figure 4.12), or because of physical CPU contention that reduces the available virtual CPU (CASE 3 in Figure 4.12). We apply the same solution regardless of which one of these two cases is causing the saturation of the virtual CPU. The solution still applies the feedback control loop, but excluding the CPU consumption of the placeholder process from the virtual CPU utilization metric, that is:

$$\text{vCPU utilization} = \frac{\text{busy}_{\text{all}} - \text{busy}_{\text{placeholder}}}{\text{busy}_{\text{all}} + \text{idle}} \quad . \tag{4.8}$$

For example, in Figure 4.14, the virtual CPU utilization is 75% if the utilization of the placeholder is not included. The virtual CPU utilization metric (i.e., the dependent variable controlled by the feedback loop) is thus not influenced by the presence of the placeholder process (which only opportunistically consumes the idle virtual CPU cycles). Therefore, in the case of traffic spikes, the proposed feedback control loop still works as in previous work [111]. In the case of physical CPU contention, the placeholder avoids the interaction between the feedback control loop (that frees the virtual CPU) and the hypervisor (that preempts the freed virtual CPU), thus allowing the feedback control loop to work correctly in this additional case.

Since CPU contention is a relatively rare event, we designed the placeholder process not to execute when there cannot be physical CPU contention. Since CPU steal time is a necessary condition (even if not sufficient, as discussed in Section 4.4.1) for physical CPU contention, the placeholder process remains idle if there is no accounted CPU steal time (CASE 1 in Figure 4.12). The placeholder process becomes active (i.e., it consumes virtual CPU cycles) once it detects that the CPU steal time has peaked (which denotes that the virtual CPU is trying to exceed a limit), and runs for a fixed amount of time T_{active}. Once T_{active} has elapsed, the placeholder process returns in the idle state. Then, the placeholder process inspects again the CPU steal time to check whether the VNF is not anymore saturating its virtual CPU. If there is still either a traffic spike or CPU contention, the placeholder process continues to be active, repeating the check later. The T_{active} should be chosen according to the expected duration of the recovery actions, such as for scaling out, hot-fixing a bug, or migrating the services to another host machine. Eventually, the virtual CPU executes again on a non-overloaded physical CPU.

4.4.3 Experimental evaluation

We performed experiments on an NFV IMS system to reproduce the problem of physical CPU contention, and to evaluate the effectiveness of overload control solutions, including both the basic and the enhanced feedback control-based approaches.

We executed experiments on a testbed based on the *Clearwater* open-source IMS system [46].

Our experimental testbed runs these components on three Dell PowerEdge servers, equipped with two 8-core 2.6 GHz Intel Xeon CPUs, connected by two Gigabit Ethernet networks, and attached to a Fiber Channel storage area network. The physical machines are managed using OpenStack (version Juno) and the KVM hypervisor (based on the Linux kernel version 3.10). Each Clearwater service is replicated in two VMs, configured with 1 virtual CPU and 4GB of RAM; each VM runs one VNF instance, and Ubuntu Linux 14.04 as guest OS. We use the *SIPp* workload generator [94] to exercise the IMS with register and call-setup requests. The IMS workload reproduces the typical message flows between subscribers, according to the SIP protocol. These flows are also adopted to test the Clearwater IMS, and the complete scenario used in our tests is available online [118].

Therefore, our workload reproduces a stressful traffic profile of 5 BHCA (i.e., Busy Hour Call Attempt) per user and 60 BHRA (i.e., Busy Hour Registration Attempt) per user. We regulate the workload intensity by varying the number of subscribers in order to reach the engineered level of the system. The engineered capacity of the experimental testbed is 40,000 subscribers, which can perform on average 660 registration requests and 55 call requests per second without SLA violations. The engineered level for the virtual CPU utilization is 75% under this reference workload.

We reproduce physical CPU contention by pinning an additional VM running a CPU-bound workload on the same physical CPU core running a critical component of the IMS. CPU pinning and CPU scheduling affinities are often considered best-practies to optimize latency-sensitive application [119]. Indeed, the scheduling affinities, can optimize memory access times in NUMA architectures and reduce the hypervisor scheduling latency. However those practies can end to increase the risk of physical CPU contention due to non optimal load balancing in case of vCPU oversubscription. Moreover setting manual CPU affinities increase the risk of contention problems due to misconfiguration by the infrastructure administrators. [120]. Thus the scenario we reproduce is representative of typical issues occurring in time-critical application running in virtual environments.

In the following, we present and discuss two groups of experiments. In the first group (Section 4.4.4), we consider a basic overload control solution, using the feedback control loop and heuristic that was introduced in Section 4.4.2. On this configuration, we reproduce overload conditions both due to traffic spikes and to physical CPU contention, in order to show that the feedback control loop can degenerate because of the vicious cycle. In the second group of experiments (Section 4.4.5), we enhance the feedback control loop with the mechanism for breaking the vicious cycle, and reproduce the same overload conditions to evaluate the proposed solution.

During the following failure scenarios, we analyze the registration attempts and the registration throughput. These quantities include both new users and retries of failed attempts. After a failure, a user starts a back-off period (uniform between 0 and 2 min) before making a new registration attempt.

4.4.4 Basic feedback control-based overload control

We deployed the basic feedback overload control in the two Clearwater VMs running the IMS P-CSCF network function, since this component is a capacity bottleneck for our deployment configuration. In a first experiment we reproduce a workload surge which is 2.5 times higher than the engineered capacity level. The experiment lasts 15 minutes and it consists of two phases: in the first phase, we gradually introduce 40,000 subscribers and wait until the workload reaches the steady state at engineered level; in the second phase, starting at second 450s, we introduce in the system 100,000 additional subscribers, causing a workload surge and the overload of P-CSCF components. Figure 4.15 shows the registration request rate and throughput of the IMS during the experiment. Before the overload phase, the average registration throughput at steady state is 624 registrations per second; during the overload phase the average throughput is 634 registrations per second, with an average CPU utilization of 73.32%. The basic overload control solution described in Section 4.4.2 (eqs. (4.6) and (4.7)) has been able to successfully protect the IMS system: it avoids service failures for already-established sessions, by correctly estimating the capacity of the system and rejecting the requests in excess with respect to the capacity, which would saturate resources and cause failures both for the initial and the new subscribers. As a result, the throughput is constant despite the traffic spike.

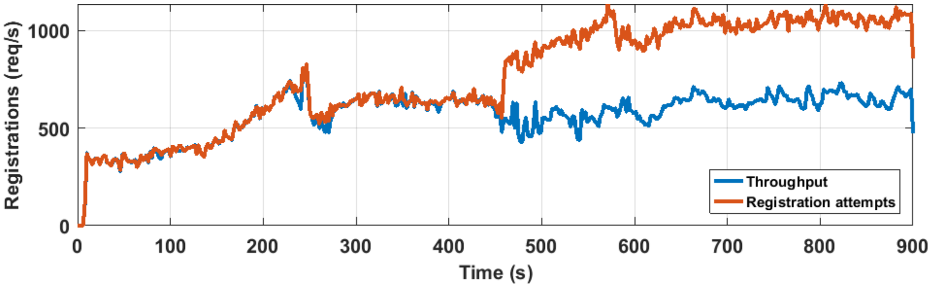


Figure 4.15. Performance of the IMS registrations during a 2.5x traffic spike (450-900s), using the basic feedback control loop.

In a second experiment, we consider again the basic control loop approach, and we reproduce an overload condition due to physical CPU contention. To this purpose, we pin the virtual CPU of the VMs running the IMS P-CSCF functions to a separate, reserved physical CPU. Then, we introduce a new VM running a CPU-bound workload (generated using the *cpuburn* tool [1]) and we pin its virtual CPU to the same physical CPU core of the IMS P-CSCF, in order to cause the contention. The experiment lasts 15 minutes (900s) and it is organized in three phases: during the first 5 minutes we generate a workload up to the engineered level; then, we activate the CPU-bound workload in the second VM to cause physical CPU contention for additional 5 minutes; finally, in the last 5 minutes of the experiment, we simulate the resolution of the CPU contention (e.g., as an effect of scaling out or migration of VMs to relieve the contention), by unpinning the virtual CPU of the CPU-bound VM. Figure 4.16 shows the registration request rate and the throughput of the system during the experiment with CPU contention.

[1] The tool can be downloaded at https://patrickmn.com/projects/cpuburn/

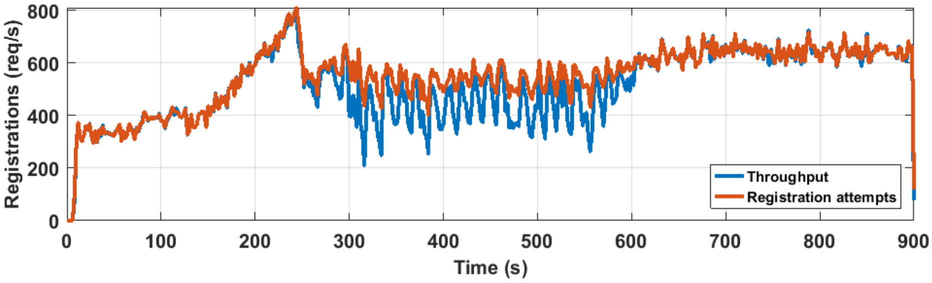


Figure 4.16. Performance of the IMS registrations during CPU contention (300-600s), using the basic feedback control loop.

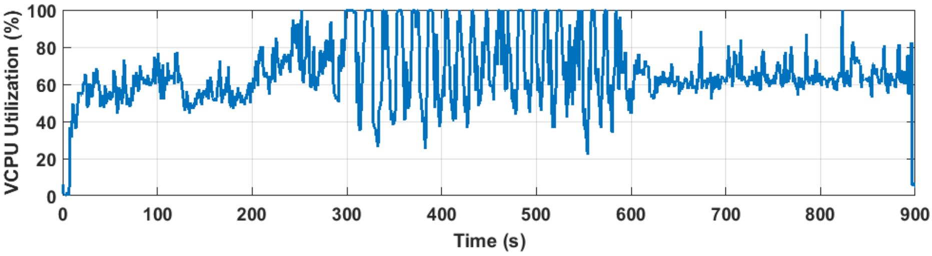


Figure 4.17. Virtual CPU utilization during CPU contention (300-600s), using the basic feedback control loop.

During the contention in the middle of the experiment, the throughput is affected by a high variability, which is a symptom that the basic control loop is unable to stabilize the load at the actual capacity of the VM. By looking at the virtual CPU usage during the experiment, showed in Figure 4.17, we noticed that as soon as we inject the CPU contention at min 5, the virtual CPU utilization raises to 100% since the hypervisor scheduler preempts physical CPU time from the virtual CPU, causing involuntary waits of the VM. As a consequence, the basic feedback control loop starts dropping part of incoming requests to reduce the virtual CPU utilization to the reference value of 75%. The Clearwater VM reduces its load and enters the vicious loop, since the CPU-bound VM takes advantage of the idle CPU time freed by the overload control mechanism. As a result, the virtual CPU utilization gradually drops down to about 20%. We also observed that the virtual CPU utilization saturates again to 100% after a period of approximately 10 seconds. This pattern is repeated periodically until the physical CPU contention is removed at minute 10, causing the high variability of CPU utilization. We found that this behavior is a side effect of the overload control mechanism, which sporadically resets the drop rate to 0 when the virtual CPU utilization becomes much lower than the reference value, thus admitting a high amount of input traffic and saturating again the virtual CPU. This high variability has a strong impact on the service latency, as further discussed in the next subsection.

4.4.5 Enhanced feedback control-based overload control

We deployed the enhanced feedback overload control strategy, and validated it by reproducing the same scenarios described in the previous subsection.

In the first experiment, after 450s, we caused a workload surge 2.5 times higher than the engineered capacity, and we evaluate the throughput of the IMS. As shown in Figure 4.18, in absence of physical CPU contention, the enhanced approach exhibits the same performance of the basic approach. Before the overload phase, the average registration throughput at steady state is 620 registrations per second while; during the overload phase the average throughput is 645 registrations per second with an average virtual CPU utilization of 74.55%. Therefore, our extension to the feedback loop does not cause any negative effect in the case of traffic spikes.

In the second experiment, we reproduced the scenario with physical CPU contention, under the same conditions of Section 4.4.4. The time series in Figure 4.19 shows the throughput of the IMS. At 5 min, we enable the CPU-bound VM. The enhanced heuristic described in Section 4.4.2 timely detected a change in the system capacity. The during the contention the average through-

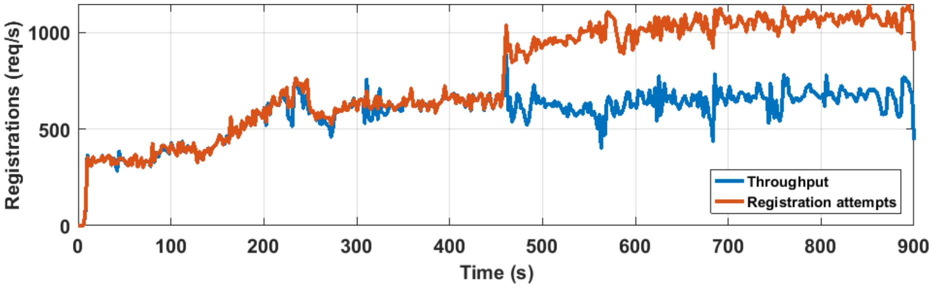


Figure 4.18. Performance of the IMS registrations during a 2.5x workload spike (450-900s), using the enhanced feedback control enabled.

put is reduced by about 32% and the system is able to complete 380 registration per second, with an average CPU consumption of 68%. Moreover, the throughput during the contention is more stable than the case with the basic feedback approach: since the placeholder process avoids the preemption of CPU time from the hypervisor, since the reference value of CPU utilization is not anymore a "moving target", thus avoiding the variations of the heuristic for capacity estimation.

If CPU contention is not properly managed, the system accepts more requests that it can actually handle with the available CPU. However, many of the accepted requests are served with a poor quality of service, and many others fail in the middle of a session (therefore, the "goodput" of the system is actually lower than the throughput). A key goal of service providers is to ensure an appropriate QoS for users that are admitted into the system, and to gracefully handle users that cannot be admitted (e.g., to notify an overload status without starting a session that cannot be assured).

It is worth noting that the throughput only appears to be higher without our enhanced control. Figure 4.21 compares the IMS throughput under CPU contention, with the basic and the enhanced feedback control loop. By looking at the throughput of the two approaches, the differences of the throughput are not significant. However, the variance of the enhanced control approach is

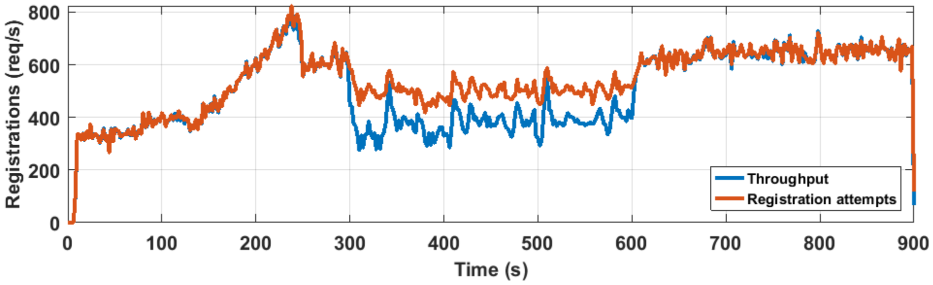


Figure 4.19. Performance of the IMS registrations during CPU contention (300-600s), using the enhanced feedback control loop.

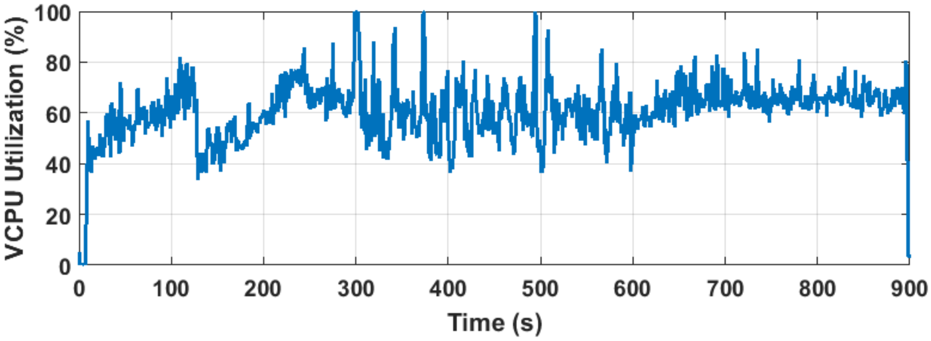
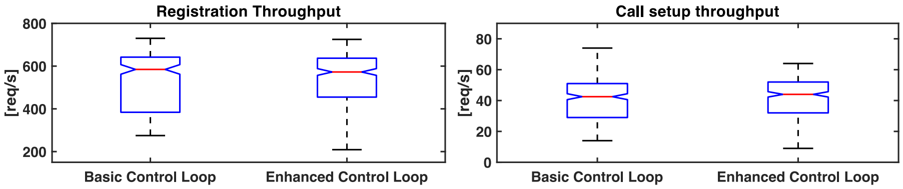


Figure 4.20. Virtual CPU utilization during CPU contention (300-600s), using the enhanced feedback control enabled.

slightly lower, for both the registration workload (Figure 4.21a) and call-setup workload (Figure 4.21b).



(a) Registration Throughput **(b)** Call-setup Throughput

Figure 4.21. IMS Registration (4.21a) and IMS Call-setup (4.21b) throughput during CPU contention, with the basic and the enhanced feedback control.

A more accurate capacity estimation has also a strong positive impact on the quality of service perceived by the IMS users in terms of *service latency*, which is a key performance indicator considered by SLAs for telecommunication systems. In particular, SLAs typically mandate latency requirements for the average (e.g., the median latency) and the worst cases (e.g., the 90th percentile of latency) [22]. In Figure 4.22, we compare the CDFs of the latency of the successful registrations, respectively under the basic and the enhanced overload control strategies, during the contention phase of the experiments. The median latency (i.e., the average case, represented by the 50th-percentile of the CDF) is up to 118.6ms for the basic approach. In the worst case, represented by the 90th-percentile, the IMS with the basic approach exhibits latencies up to 369.9ms. These latency values are close, and even exceed the SLA objectives typically adopted for IMS systems (e.g., 150ms and 250ms respectively for the 50th and 90th percentiles) [5, 121]. Instead, the proposed approach significantly improves the quality of service, by achieving a service latency up to 28.5ms and 106.2ms respectively for the 50th and 90th percentiles.

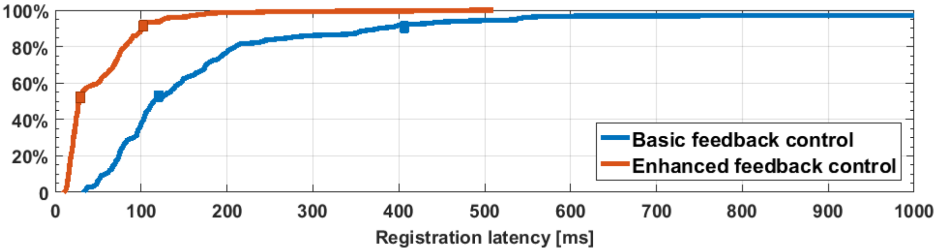


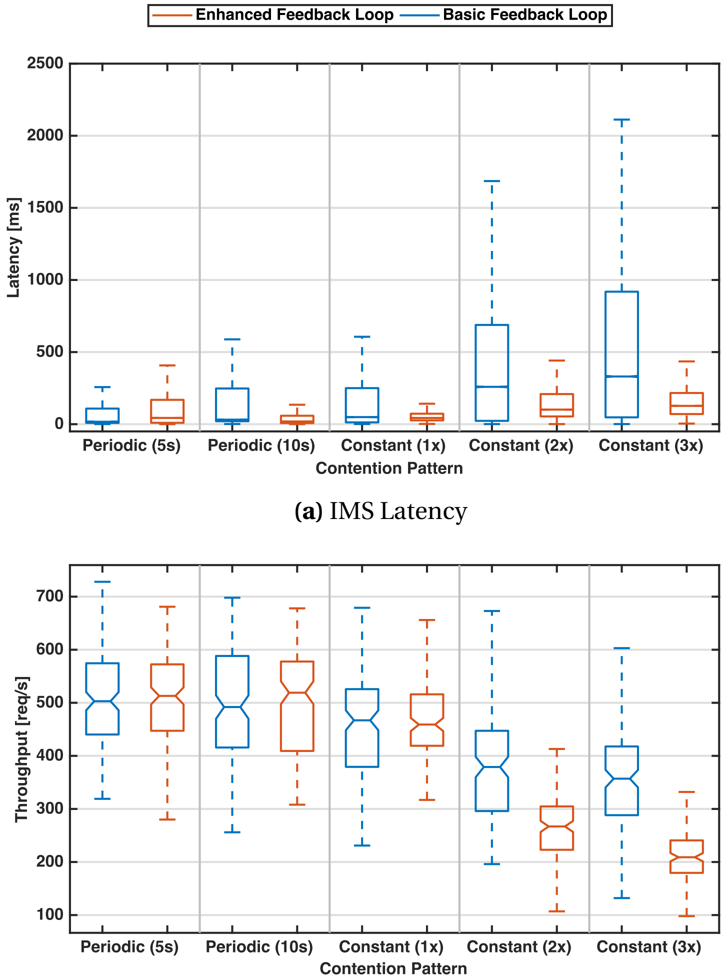
Figure 4.22. Cumulative distribution of registration latency, with the basic (red line) and the enhanced (blue line) feedback control.

4.4.6 Performance evaluation under different contention patterns

In the following, we present another group of experiments, to assess the performance of the basic and the enhanced feedback loops in response to different CPU contention patterns. The purpose of this analysis is to identify which scenarios will benefit the most from the proposed solution. We vary the *intensity* of the CPU contention, and the *duration* of CPU contention periods.

- **Intensity**. The intensity of contention is determined by the amount of competing virtual machines that are deployed on the same physical machine. The intensity of contention can impact on the variability of CPU utilization by the virtual machine (e.g., the amplitude of swings in CPU utilization metrics), with side effects on the overload control loops. Therefore, we performed additional experiments where we vary the intensity of contention between 1x (i.e., 50% available CPU time due to the CPU contention with one additional VM) and 3x (i.e., 25% available CPU time due to the CPU contention with three other VMs).
- **Duration**. The duration of contention is determined by the overlap over time of CPU-bound activities on several virtual machines. If contention periods are long (e.g., due to a configuration error with persistent effects), then the overload control algorithm can eventually converge to a stable condition; instead, if contention periods are short and intermittent (e.g., due to transient high CPU usage by background tasks in the VMs), the overload control algorithm may exhibit unstable behavior and poor performance. Therefore, in addition to the previous experiments (where the CPU contention is constant for a relatively long period), we perform more experiments with short, periodic contention periods, where the periods last respectively for *5s* and *10s.*

We applied these conditions both on the basic and on the enhanced feedback loop solutions. Each experiment lasts 15 minutes (900s) and it is organized in three phases: during the first 5 minutes we generate a workload up to the engineered level; then, for 5 more minutes, we force physical CPU contention (either periodically or constantly, depending on the *duration* as dis-



(a) IMS Latency

(b) IMS Throughput

Figure 4.23. IMS Throughput (4.23b) and IMS Latency (4.23a) under different CPU contention patterns, with the basic and the enhanced feedback control.

cussed above), by activating the CPU-bound workload in the additional VMs (between one and three VMs, depending on the *intensity* as discussed above); finally, in the last 5 minutes of the experiment, we simulate the resolution of the CPU contention, by unpinning the virtual CPU of the CPU-bound VMs, as an effect of scaling out or migration of VMs to relieve the contention.

Figure 4.23 summarizes the performance of the IMS (latency and throughput) during these additional scenarios, with both the basic (blue boxes) and the enhanced (red boxes) feedback loop strategies. When the contention period is very short (5s) there are no significant differences between the two solutions. This scenario represents the most unfavourable condition for our enhanced solution, since the control feedback is based on a sampling window of 5 seconds, and thus the proposed solution is unable to provide any improvement. The percentage of requests violating the latency goal of 250ms is 9.6% for the basic approach and 9.8% with the other. In both cases, the SLA goal of 90%-percentile is not violated.

Starting with a period of 10s up to constant patterns, the enhanced feedback loop shows a significant improvement of latency compared to the basic feedback loop. In the case of a periodic contention of *10s*, only the 0.1% of the requests experiences latency higher than 250ms, in contrast to the basic approach in which the 20.1% of the requests were served with a latency higher than the requirement, thus violating the SLA goal.

With a constant contention pattern at 1x intensity, the average available CPU capacity is reduced to 50%, since the CPU is contended between 2 VMs. As discussed in the previous section, there are no significant differences in the IMS throughput between the two approaches, but the proposed approach shows significant reduction in latency: by using the enhanced approach the percentage of requests violating the 90%-percentile requirement decreases from 17.9% to 0.2%.

This behavior is exacerbated by higher contention intensities (2x, 3x). In these cases, the basic overload control solution is unable to accurately estimate the available CPU capacity, due to the wider swings in CPU utilization metrics; therefore, it degenerates by accepting more requests than the actual capacity of the IMS system. The result is a significant increase of IMS latency in the basic feedback loop. With a constant CPU contention at 2x intensity, the available CPU capacity of the VNF is reduced to 33% on average, since the CPU time is contended with 2 additional VMs. In this case, more than the 42% of the requests violates the 90%-percentile latency requirement, in contrast to the 8.1%, when using the enhanced approach. With CPU contention at 3x intensity, with only 25% of the CPU time is available to the VNF. By using the enhanced solution, the number of requests violating the 90%-percentile latency requirement drops from 48% to 9%, thus achieving the SLA requirement.

It is interesting to note that, under the higher intensities of CPU contention, the average throughput with the basic approach is higher than the enhanced approach (e.g., 380 req/s versus 270 req/s in the case of 2x intensity). The (apparently) better throughput comes at the cost of a poor quality of service, since the IMS is processing a volume of requests which is higher than its capacity. The result is that the IMS takes a long time to serve many of these requests, thus violating the latency requirement. Instead, the enhanced solution only lets in the IMS the subset of requests than can be processed with adequate quality of service: this is a desirable effect of throttling, which is intended to drop the traffic in excess to the system. This result points out that the proposed feedback solution is best suited for those applications (such as the IMS, and NFV in general) where latency and throughput are both important SLA goals. If the proposed solution is not deployed, the throttling mechanism degenerates and lets in the IMS too much traffic, thus favoring throughput at the expense of latency.

Chapter 5

Managing the overload of stateful multi-tier network functions

Modern multi-tier architectures achieve massive scalability by balancing the load on thousand stateless application nodes, and leveraging highly distributed NoSQL datastores as persistence tier, in order to achieve the required levels of performance and reliability. This Chapter revisits overload problems that affect these architectures, and it proposes *DRACO*, a novel autonomic solution that addresses overloads arising in any tier of the system. When overload occurs in the inner tiers, overload control must be aware of the dependencies between the application and the storage resources, and, in the case of unbalanced overload conditions (such as hot-spot resources), it should only drop application requests that map to resources on overloaded storage nodes. DRACO is a fine-grained admission control solution, which has been designed to dynamically discover such resource dependencies and assess the current capacity of individual nodes, in order to mitigate overloads while achieving a high resource utilization. DRACO is evaluated on two case studies: a Distributed Fileserver, which is very sensitive to problems of data consistency and hot-spots, and a virtualized IP Multimedia Subsystem, which requires carrier-grade levels of performance and availability.

5.1 The problem of unbalanced overloads in multi-tier systems

At a higher level, a modern multi-tier architecture is composed by two kinds of tiers: *stateless*, such as application ones, and *stateful*, such as datastores. A typical interaction between these elements is showed in Figure 5.1. Clients interact with the system by issuing *service requests* towards one of the nodes of the application tier. These service requests are uniformly distributed to application nodes according to a load balancing mechanism (e.g., round-robin selection from a pool of IP addresses, using DNS). Since application nodes are stateless, they will forward *data requests* to a datastore cluster in order to retrieve data needed to process the service request; moreover, they will issue data requests to update the datastore to reflect the state of the distributed application. For example, the storage tier can be used to hold data records for user authentication, billing, tracking long-term sessions, etc., and even to hold large binary blobs, such as for multimedia and file sharing applications.

In general, a single service request causes one or more data requests to the datastore tier, either sequentially or concurrently. Differently from the service requests, data requests need to be directed to the specific storage nodes that manage the resources required by the user. The location of resources is typically identified using *consistent hashing* [34], which computes a lightweight, deterministic function to map a service request to storage nodes. With this approach, the dependencies among application nodes and datastore nodes are dynamically generated based on the content of service requests from the clients and, thus, they are not known a-priori.

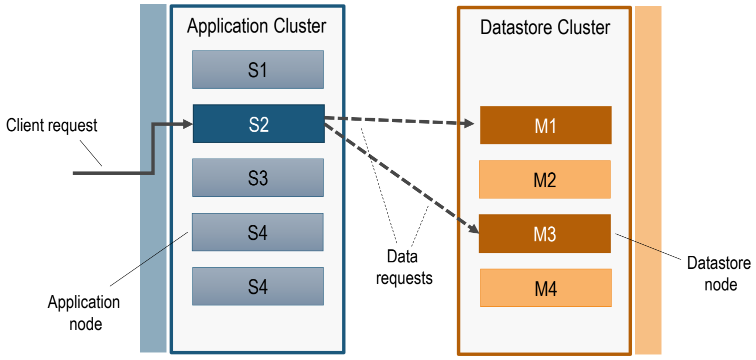


Figure 5.1. The typical multi-cluster architecture

An overload condition occurs when a system has insufficient resources to serve the incoming requests. This condition happens when the current workload hits a bottleneck in one of its components, which limits the capacity of the whole system. In principle, overloads can be avoided with an appropriate capacity planning, by identifying the potential bottlenecks and characterizing their performance. Unfortunately, in a multi-tier architecture, a bottleneck can dynamically shift from a tier to another according to changes in workload patterns. Moreover, especially in large-scale systems, new (and unexpected) bottlenecks can arise from specific nodes of a tier due to server heterogeneity, software bugs, maintenance tasks and misconfigurations. As result, in practice, preventing overloads in such systems is a challenging task.

A bottleneck in the application tier is the easiest case to manage. Application nodes are stateless and independent from each other by design, and requests can be dispatched to any node without restrictions. These properties make possible a uniform load balancing among nodes in the tier. Therefore, an overload occurs when the available capacity of the tier as a whole is not enough to serve the incoming requests. In this case, the existing solutions, such as throttling and scaling out the tier, can efficiently mitigate the overload. Throttling performed in the application tier can reject requests that go beyond the available capacity of the tier, while admitted requests can be served at peak performance; moreover, since nodes are independent from each other, scaling out the tier is relatively easy and, thanks to load balancing, leads to an increase of the available capacity of the tier as a whole. In a similar way, the communication between two or more stateless application tiers follows the same principles.

Overload conditions are more challenging when caused by the interaction between stateless and stateful tiers. Indeed, when some components of the application architecture are stateful (e.g., a datastore), a typical problem is represented by *hot-spots*: when users access a subset of resources much more frequently than others (for example, multimedia content or application that suddenly becomes popular on the web), the load on the stateful nodes that manage these resources will be higher. When the load on this subset of nodes exceeds their capacity, the application services also become prone to failures. A symptom of an hot-spot is a highly skewed distribution of the requests to the nodes. Figure 5.2 shows an example of skewed request distribution, causing an overload in a group of storage nodes.

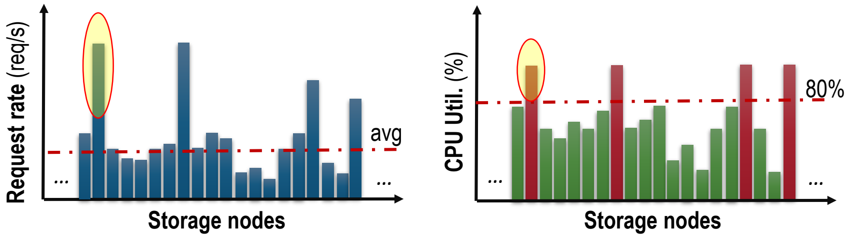


Figure 5.2. Distribution of the request rate and the CPU utilization across the data-tier nodes

The effect of the hot-spot is an unbalanced load among the storage nodes. This problem is difficult to address since it is not simply caused by the configuration of the nodes or by the scale of the tier, but it depends on which services and resources are requested by the external users. Since the workload profile is not known a-priori, it is difficult to manage the capacity of the stateful tier. Due to the hot-spot problem, every stateful node can be a potential bottleneck for the entire cluster, even when there is available capacity in the remaining nodes.

Figure 5.3 shows two external clients that make requests to the application tier. In order to complete these requests, the two application nodes (i.e., S1 and S4) need to access the same resource in the node M3. Thus, the node M3 has less resources available for subsequent requests, and can become overloaded more quickly than the other nodes in the storage tier.

Redistributing the resources and scaling the stateful tiers can alleviate the problem as a long-term solution, but they are not effective in the short-term. Since these strategies require time and resources (e.g., scaling-out with new virtual machines can take up to several minutes [23, 24]), the system can remain prone to service failures (e.g., SLA violations) for a significant amount of time. Therefore, in order to mitigate the unbalanced overload in the short term, the system needs to reduce the volume of requests for hot-spot resources.

However, the storage tier cannot simply discard the requests, since this would either violate data consistency (e.g., ACID properties for transactions that span over several storage nodes); or, it would trigger a complex and wasteful roll-back of the transaction. For example, in Figure 5.3 the Client 2 makes a service request that updates resources in the datastore nodes M3 and M4. If the node M3 rejects the update while the node M4 completes the update, the datastore may be left in an inconsistent state. For the above reasons, an overload control mechanism should prevent the hot-spot by rejecting service requests at the application tier, *before* they enter into the multi-tier system.

It is worth noting that even if the load from external users is balanced, overloads can still affect the stateful nodes. The use of hashing schemes (such as consistent hashing) ensures a uniform distribution of the resources among

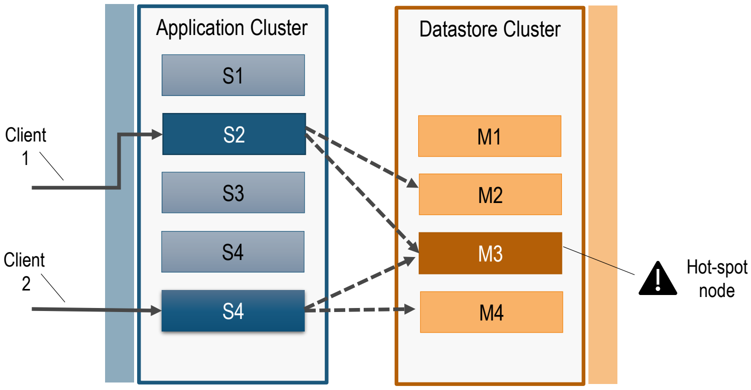


Figure 5.3. The overload scenario caused by hot-spot resources

the datastore nodes. However, these nodes may have different configurations; they can be deployed on physical machines with different characteristics; or they can be shared among several services from different tenants. As a consequence, the nodes of the distributed datastore can exhibit a different capacity. This means that some nodes can become overloaded more quickly than the others.

Figure 5.4 shows an example in which the requests to the stateful tier are balanced across the M1, M2 and M3 nodes. However, if the node M3 is misconfigured with less capacity than the other nodes, it becomes overloaded, and the application tier experiences service failures even though the nodes M1 and M2 still have available capacity.

As discussed for the case of hot-spots, redistributing resources can mitigate the problem only on the long-term, since resource migration is an slow operation. In order to avoid service failures, the load on the storage tier should not be balanced uniformly across nodes, but an overload control mechanism should block new requests that would put more load on the saturated storage nodes. Moreover, to maintain the consistency of the datastore, the overload control should reject requests at the application tier, before they enter into

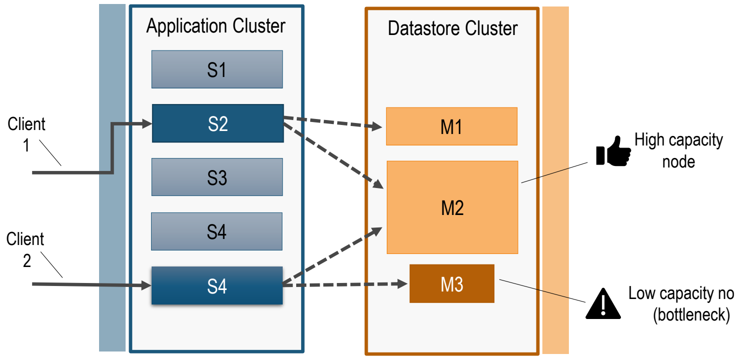


Figure 5.4. The overload scenario caused by unequal node configurations.

the system.

Finally, even when the nodes have the same resources, overloads can still occur due to variations of the capacity. These variations can be due to overlaps with background or periodic tasks, or due to resource “hogs” caused by software bugs. The effect of this scenario is that a storage node can be temporarily overloaded even though it has enough resources to handle the requests.

Figure 5.5 shows an example of this problem. In this scenario, all nodes are configured with the same capacity and the datastore is configured to distributed resources uniformly across its nodes. The requests from the two clients are balanced across the nodes. However, part of the requests on behalf of the Client 2 fail since the node M3 is temporary overloaded by a competing task that consumes its available capacity.

Since these capacity variations are unexpected, the multi-tier system should be able to dynamically adapt. However, even in this scenario redistributing the resources cannot solve the problem in the short term. Therefore, an overload control should mitigate the unbalanced overload by rejecting user requests before they reach the overloaded storage node.

In summary, overload control should meet the following requirements in order to mitigate unbalanced overloads in all of the scenarios discussed above:

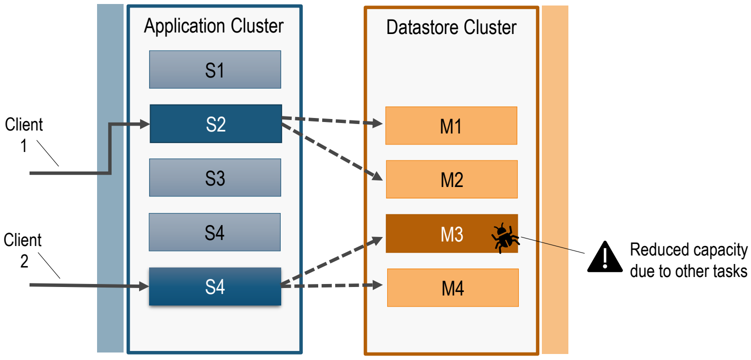


Figure 5.5. The overload scenario caused by a transient reduced capacity

1. *The capacity of storage nodes must be dynamically monitored, to detect hot-spots and transient capacity variations that may be caused by resource hogs.*

2. *The application requests in excess should be rejected before they are processed by the application, in order not to perform partial operations on the storage tier, which would violate data consistency or cause a waste of resources.*

3. *In order to use resources efficiently, the overload control solution should only reject service requests that access to overloaded storage nodes. Instead, service requests that do not access these nodes should be allowed in the system.*

5.2 The proposed solution

The proposed solution is an enhanced overload control approach that is aware of data dependencies in multi-tier systems. The main novel feature of *DRACO* is the ability to mitigate unbalanced overload conditions that arise from a subset of nodes in the storage tier, and at the same time to achieve a high utilization of the non-overloaded parts of the storage tier. The driving idea to achieve this goal is to opportunistically take advantage of knowledge of the application logic, in order to map service requests to the storage nodes needed to serve that request; and to only admit into the system a selected subset of service requests, by rejecting the ones that would attempt to get data from storage nodes that are already overloaded.

The solution adopts a distributed architecture, in order to scale well with the size of the tiers. Moreover, the solution is designed to filter traffic at the application tier; that is, it avoids to filter the traffic at the storage tier, which would cause inconsistencies between the application and storage tiers, and among the nodes of the storage tier. In the following, we present the solution by introducing its general architecture and components (Figure 5.6). In sections 5.3 and 5.4, we will discuss more in detail the implementation of this general architecture in the context of two case studies.

The main part of the solution is the *Distributed Memory* block. This component keeps track of the *location of resources* across storage nodes, and of the *residual capacity* of the storage nodes. The residual capacity is an estimate of *the number of data requests that a storage node can serve*, which is computed by algorithm (described more in detail later) according to the number of data requests previously served by storage nodes, and the corresponding utilization of critical resources (in particular, the CPU consumption) when these data requests were served. This information is periodically collected by a *Capacity Monitoring* component distributed on the nodes of the datastore cluster.

The residual capacity gives indication of how many data requests are in excess in the storage tier, and it used to identify which service requests should be rejected at the application tier by the *Distributed Admission Control* component. The admission decision is done by inspecting the service request, and by checking if there is enough residual capacity in the current application node and in all of the storage nodes that are needed to process the service request.

In summary, the components of the *DRACO* are:

- The *Distributed Memory*, which stores the information about the location of resources and the residual capacity of the storage nodes.
- The *Distributed Admission Control*, which acts as a tunnel between the users and the application processes, and which decides if an incoming service request should be accepted or rejected, according to the information produced by a *Resource Location Discovery* phase aimed to identify the required datastore resources and their locations.
- The *Distributed Capacity Monitoring*, which is in charge of collecting resource consumption metrics from storage nodes. These metrics are used to dynamically estimate the residual capacity of storage nodes (e.g., in terms of number of read/write accesses that can be performed on the storage node).

5.2.1 The Distributed Memory component

The *Distributed Memory* is an additional datastore that handles the following two kinds of data (Figure 5.6):

- **Node Capacity Status**: The residual capacity of every storage node, in terms of number of requests the node can accept in current time window. This information is periodically updated by the Capacity Monitoring block. At the beginning of a new time window, this block estimates the residual capacity according to the load of the storage nodes in the previous windows. Moreover, this information is read by the Admission Control block in order to check if there is enough available capacity in the required nodes, and will be also decremented upon the acceptance of a service request.

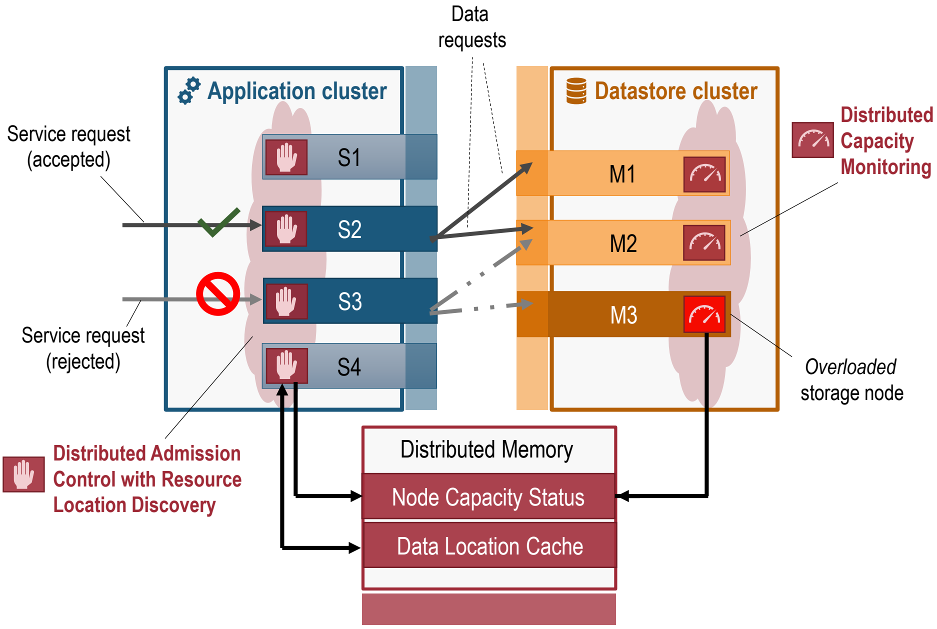

Data requests
Application cluster
Datastore cluster
Service request (accepted)
Service request (rejected)
S1
S2
S3
S4
M1
M2
M3
Distributed Capacity Monitoring
Overloaded storage node
Distributed Admission Control with Resource Location Discovery
Distributed Memory
Node Capacity Status
Data Location Cache

Figure 5.6. Overview of *DRACO* architecture (red blocks).

- **Data Location Cache**: The location of the resources accessed by service requests. This information is added, retrieved and updated during the *Resource Location Discovery* phase of the Distributed Admission Control.

The *Distributed Memory* stores this information as key-value pairs. The value for the Node Capacity data is an integer representing the number of requests that can be served by the storage node in the current time window. This number is associated to a key that represents the node (e.g., using its hostname or IP address). The Data Location Cache stores a key-value pair for each service request. The value of an entry is an array of integer values, with one element for each storage node. These values represent the number of requests to perform on each storage node in order to process the service request. This information is associated with a key, which is application-dependent, that represents a specific service request.

We leverage existing datastore technology for this part of the overload control solution. In particular, using a distributed datastore (such as Memcached) allows the solution to handle data from a large number of nodes in the clusters, and simplifies the collection and the distribution of capacity monitoring data across the nodes. The *Distributed Memory* can be deployed either on any of the existing tiers, or in a dedicated tier, for example by introducing a stand alone group of VMs running the datastore. To avoid performance bottlenecks and to achieve scalability, we create a fixed pool of persistent connections between each node and the Distributed Memory. Thus, the number of connections to the cache only grows linearly with the amount of nodes in the cluster. The solution avoids any direct communication between pairs of nodes in the tiers. Moreover, the capacity of the Distributed Memory can scale linearly with the number of its nodes, by splitting the information about resources across several nodes. In section 5.5 we discuss in detail about the capacity planning and scalability of the *Distributed Memory*.

The *Distributed Memory* block stores the location of the resources accessed by previous service requests. The use of such distributed cache improves the performance of the overload control solution, especially in the case of hot-spot resources, since the information on hot-spot resources (which are repeatedly accessed in a short amount of time) is likely already cached by this block. However, this cache is not always necessary. When the application can retrieve the location of resources solely from the request message (e.g., by applying *consistent hashing* on the fields of a service request), the Admission Control block can perform the same computation and find the location of the resources. In this way, caching resource locations is not strictly needed. It is useful to still have a distributed cache if the application needs to access to a large number of resources per service request, and when the computation is expensive. On the contrary, if service requests involve only few resources and there are no hot-spot resources (e.g., as in the case of an IMS scenario discussed in Section 5.4), it can be advantageous to compute the location directly, avoiding to use the cache.

5.2.2 The Distributed Capacity Monitoring

The *Distributed Capacity Monitoring* component (Figure 5.7) is deployed within every storage node, and it is in charge of dynamically estimating the available capacity of the storage node. This block will collect information about resource utilization from the OS or from the hypervisor on the storage node. In particular, we focus the discussion on the case where each monitoring block estimates the capacity of a storage nodes with respect to its *CPU utilization* (i.e., in terms of percentage of busy CPU cycles per unit of time), since the CPU is often the resource most prone to become a bottleneck [122]: for example, in the context of NFV [123], industry-standard COTS CPUs are adopted to process high volumes of network traffic. In addition to CPU utilization, the proposed approach can be easily generalized to be applied on memory, network, and disk bandwidth utilization.

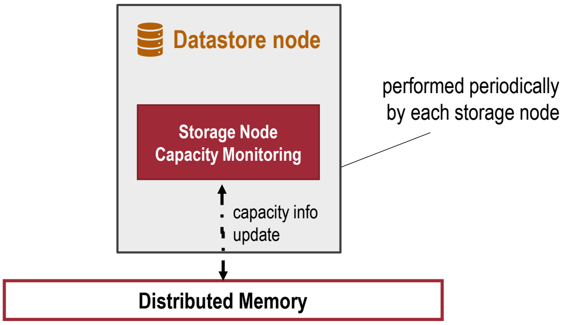

Datastore node
performed periodically by each storage node
Storage Node Capacity Monitoring
capacity info update
Distributed Memory

Figure 5.7. The Capacity Monitoring block

This block periodically estimates the residual capacity, in terms of number of requests that the node can serve in a time window. The time window is meant to be short (e.g., in the order of few seconds), in order to quickly adapt the solution to variations of the load condition of the storage node. In addition to the CPU utilization, the Capacity Monitoring block uses the number of data requests that have been served by the storage node in the last time window, which is recorded by the Admission Control block when a service request is accepted. The residual capacity is estimated according to the following equation:

$$\text{residual capacity} = \frac{\#\text{ data requests}}{\texttt{CPU}_{\texttt{used}}\%} \cdot \texttt{CPU}_{\text{reference}}\% \quad . \tag{5.1}$$

In this equation, the first factor estimates the cost of an individual data request, in terms of CPU cycles, by dividing the number of data requests in the last time window with the average CPU utilization during the same period. This approximation is simple but still accurate enough in the context of multi-tier systems, since the complexity of an individual data request is relatively low in the case modern datastores. In the case of older types of storage systems (e.g., based on a traditional SQL DBMS), the cost of an individual data request would depend on the types of SQL queries performed by the application, and it should be estimated using a more complex cost model [124]. Since we focus this work on modern datastores, we leave out of scope the analysis of other cost models.

The second factor in the equation represents the *reference CPU budget*, beyond which the storage node is considered saturated. This value represents a "factor of safety" for CPU utilization, within which the storage server is designed to perform well (e.g., without performance disruptions), while leaving a small amount of residual CPU bandwidth to handle occasional load fluctuations. We base our algorithm around a reference value since setting a reference is a frequent practice among system administrators (e.g., for monitoring and troubleshooting purposes). For example, when testing the capacity of a system (e.g., by imposing a representative workload), the system administrator may want to check that the CPU utilization is below a reference value (e.g., 90%), and that a good quality of service can be provided under these conditions.

In Eqn. (5.1), these two factors are multiplied to get the total amount of data requests that can be correctly served by the storage node. The capacity value is then updated in the distributed cache. The Algorithm4 is executed periodically within the Capacity Monitoring block deployed in each datastore node. It updates the available capacity budget for the local storage node in the Distributed Memory component, according to the request rate and the corresponding CPU utilization measured during the last period.

Algorithm 4: Capacity Monitoring algorithm

```
// Periodically on the "node capacity status" block
begin
    // Get CPU utilization and the number of storage
    requests // that arrived since the previous update
    $CPU_i$ = get_CPU_utilization($node_i$)
    $\#requests_i$ = get_served_storage_requests($node_i$)

    // Compute the capacity budget of the storage node
    (see eq.  (5.1))
    $C(i)$ = compute_capacity_budget($node_i, CPU_i, \#requests_i$)
    node_capacity_status.update_capacity_budget($node_i$, C(i))
```

5.2.3 The Distributed Admission Control

Figure 5.8 shows the internal organization of the *Distributed Admission Control.* A Data Location Discovery step is performed when a service request is received by the application tier, and before it is processed by the application tier. This component produces a list of the resources needed by an application request, and of the storage nodes where these resources are located. The Admission Logic uses this list in the decision process. The implementation of this component depends on the specific application, and it is meant to be tailored by the application programmer. From a general point of view, the Data Location Discovery parses service requests by looking for information that uniquely identifies the data needed by service requests, such as the user identity, the session identifier, the name of the resource(s) involved in the service request (such as, the identifier of application records, or the name of multimedia content), the type of operation to be performed on the resource, and similar information. This information is then used to query the distributed cache to find the location of resources in the storage tier. The main assumption of the proposed solution is that service requests hold such information, and that it allows to establish the mapping with storage nodes. This assumption holds in practice for many applications: since application nodes are stateless, the service request message includes all the information needed by the application logic to access the datastore. We will further discuss this aspect in the context of two case studies (sections 5.3 and 5.4).

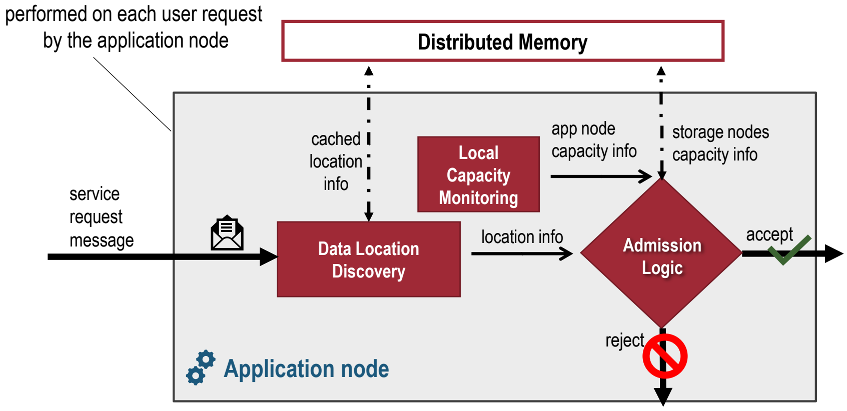


Figure 5.8. The Admission Control process

If the service request involves resources that may have already been accessed in the past (such as, a request to retrieve a resource for the user), the Data Location Discovery block checks if there is an entry for the service request in a distributed cache, and retrieves information on the resources from there. It is possible that an entry does not exist yet (for example, the service request comes from a new user); in this case, the Data Location Discovery block uses the information extracted from the request to find the location of the required resources in the storage tier, then it updates the cache.

More specifically, in the case of a storage tier with a NoSQL datastore, the Data Location Discovery block computes the same hash function that is computed by the application to resolve the location of a resource on the datastore, according to the technique of consistent hashing (as previously discussed in section 5.1). In other cases, the Data Location Discovery block obtains the location of a resource by retrieving resource metadata. In Figure 5.9, we refer to this computation as the *location function*, which maps the information from a request (i.e., the “key” of an entry in the datastore) with the corresponding resource. The figure provides two examples of location functions in the context of the two case studies that will be presented in sections 5.3 and 5.4. In the first example, the resource is represented by a file block; the key is represented by the combination of the filename and of the numeric identifier of the block; and the output of the function is the storage node with the block. In the second example, the resource is represented by a record with user information; the key is represented by a combination of the username and the SIP URI; and the output is the storage node with the record. In general, the output of this computation is the set of nodes that need to be accessed by the service request (e.g., several nodes in the case that a service request involves several resources, or that the resources are replicated across several nodes for fault-tolerance).

One approach to implement the Data Location Discovery block is to integrate it in the Admission Control block, and to execute it in a separate process

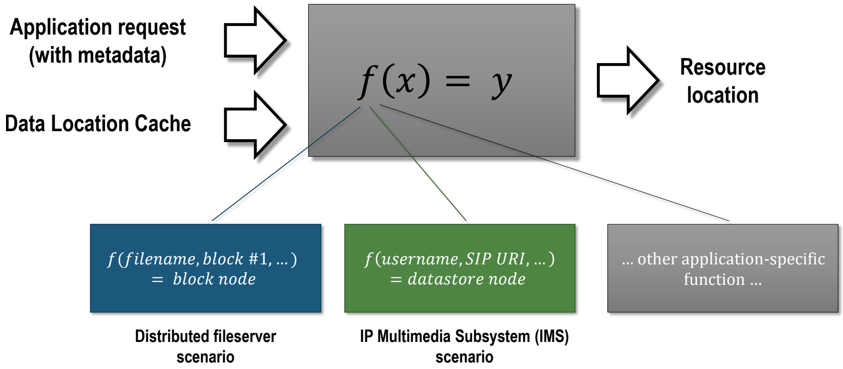

Application request (with metadata)
Data Location Cache
$f(x) = y$
Resource location
$f(filename, block\ \#1, \ldots) = block\ node$
$f(username, SIP\ URI, \ldots) = datastore\ node$
... other application-specific function ...
Distributed fileserver scenario
IP Multimedia Subsystem (IMS) scenario

Figure 5.9. The location function maps application requests to their location in the storage tier

that tunnels the traffic from clients to the application tier. In this case, application developers have to supply a small module (to be linked with the data location discovery block before deployment) that implements the location function. The Data Location Discovery block calls this module, by passing in input a incoming service request. In the module, the developers provide code to parse the service request and to extract information for mapping the request to storage nodes. Then, the information is used to query the distributed cache, or to compute the location function as discussed above. Since both the information extraction and the location function are simple operations, their overhead on service requests is expected to be negligible; this overhead is further analyzed in the experimental part of this work. An alternate approach is to implement these blocks as a library linked to the application, in the case that developers need to further reduce this overhead and are willing to introduce small modifications to their application. The application would call the library API when there is an incoming service request, by querying the *Distributed Admission Control* component to check if the request should be processed; since the application already computes the location function to locate the resources, it is possible to reuse the results of this computation in the Admission Control and Data Location Discovery blocks.

The Algorithm 5 describes the steps performed by the *Distributed Admission Control* on every service request, before it is processed by the application, in cooperation with the other components. The Admission Control block first checks if local application nodes has enough available capacity to process the service request. The local capacity budget C_{local} is computed similarly to the Distributed Capacity Monitoring discussed in Section 5.1 for the storage nodes: The available capacity is periodically estimated according Eqn. 5.1 by considering the number of service requests accepted in the last period and the CPU consumption of the local node. Differently from the Capacity Monitoring in storage tier, this information is not exported to the other nodes through the Distributed Memory component, since it will be used to prevent the overload of the local application node.

If the local node has enough available capacity to process the incoming service request, the Algorithm 5 gets the location array $L(1..n)$ from the Data Location Discovery process: the i-th component of this array represents the number of data requests that will be directed to the storage node i. Moreover, the Admission Control block retrieves from the distributed cache the capacity array $C(1..n)$: the i-th element of this array represents the number of data requests that the storage node i can accept in the current time window without saturating its capacity. This node capacity status is updated on the distributed cache by the Capacity Monitoring block.

The algorithm compares, for each storage node i, the residual capacity of the node with the number of data requests for the node. When $L(i) > 0$, there is at least one resource on the i-th data node required to complete the current service request. If there is at least one storage node in which the residual capacity is not sufficient to process the data requests (i.e., $C(i) - L(i) < 0$), the algorithm decides to reject the service request. Otherwise, it will accept the service request. In this case, the Algorithm 5 discounts the number of data requests towards node i from the residual capacity of the node i, and it will update the residual capacity of the node in the distributed cache. Moreover, the algorithm updates the residual capacity for the local application node to reserve a capacity budget (i.e., 1) for the accepted service request. Since in many applications, some service requests may require more capacity than the others, the algorithm can be easily adapted to account a weighed local budget capacity, by subtracting to C_{local} a quantity $B_{req_type} > 0$, depending on the request type. For example, in the IMS case study, a SIP Register service request requires less application tier CPU resources than a SIP Invite request. Moreover, this approach is robust due to a control loop feedback: if the estimated capacity is lower (or greater) than the actual capacity of the node, the CPU consumption of the node will decrease (or increase) respectively. When this happens, in the next control period the estimated capacity will be higher (or lower) according to the heuristic in eq. (5.1) to compensate the error of the previous control period.

5.3 The Distributed Fileserver Case Study

We analyze the overload control solution in the context of a distributed file system service, based on the *Memcached* datastore. This service includes three tiers: a frontend tier, based on *HAproxy*, which performs load balancing among applications nodes; an application tier, composed of a cluster of web application nodes, which accepts users' requests, and translates them to requests towards the data tier; the datastore tier, which is composed of a pool of Memcached nodes. Figure 5.10 shows how these tiers are related. The web application has been developed by us in order to reflect the architecture of a proprietary distributed fileserver from our industrial project partners, and to reproduce the unbalanced overload scenarios that they experienced in the context of this application.

Algorithm 5: Admission control algorithm

```
// Part 1: Arrival of an incoming service request (to
be accepted or rejected)
Data: $app\_request$: the incoming service request
begin
    // Check if there is enough capacity on the current
    node
    $C_{local}$ = get_local_capacity_budget()
    if $C_{local} = 0$ then
        REJECT the request
    $M$ = get_metadata($app\_request$)
    $L$ = data_location_cache.get_location($M$)
    foreach storage node $i$ in $L$ do
        // Get the current capacity of the $i$-th node
        $C(i)$ = node_capacity_status.get_capacity_budget($node_i$)
        // Check if there is enough capacity to perform
        // $L(i)$ accesses to the storage node
        if $C(i) - L(i) < 0$ then
            REJECT the request
    // Decrement the capacity budget for all storage
    nodes
    foreach storage node $i$ in $L$ do
        node_capacity_status.update_capacity_budget($node_i$, C(i) -
        L(i))
    // Decrement the local capacity budget
    update_local_capacity_budget($C_{local}$ - 1)
    ACCEPT the request
```

The user can request 4 types of operations on the system: (1) registration, (2) upload of a file, (3) download of a file, and (4) de-registration. Clients can select any instance of the HAproxy by querying a DNS server (*bind9* in our

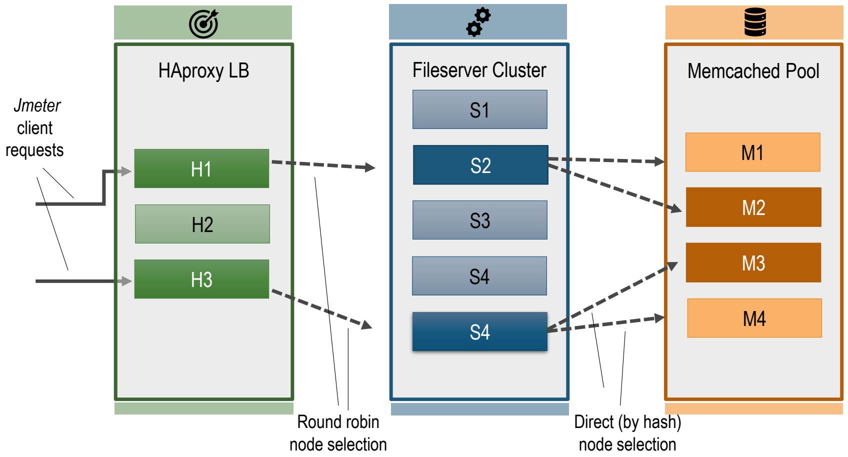


Figure 5.10. The architecture of the distributed fileserver casestudy

setup). The application cluster is stateless: no session state is stored in the web server. For example, it is possible to send a register request to the server S1 and an upload request to the server S2. The stateless web application stores the data in the nodes of Memcached key-value store cluster. The four operations that a client can perform are implemented in a C application based on *libevent* library to perform asynchronous networking communication with the clients, and *libmemcached* to communicate with the datastore tier.

The register is a “set” operation that stores user account information on a Memcached server (such as the username and the last access time) under a specific key, while the unregister is a “delete” operation that removes the key-value pair from the server. To perform an upload request, the client sends a file through an HTTP POST request message that includes the “username”, the “filename” and the file content. Then, the application divides the file content into chunks of equal size (1MB) and stores the chunks into data nodes. The application uses the string key `"username$filename$1"` to identify the first chunk, and the string `"username$filename$n"` to identify the chunk *n*. Then, sequentially for each chunk, the application computes the hash on the key `MD5("username$filename$1)` to identify the location of a data node and send the chunk to that node. Finally, the application uses the key `"username$filename"` to store the entire file size, which is obtained from the `Content-Length` HTTP header. The upload function takes the username, the filename and the content as parameters, and uses a **set-Multi** function to store an array of key-value pairs on the servers. To serve a download request, the application uses the same strategy: it extracts the "username" and the "filename" from the request message. Then, it gets the current file length by querying the key `"username$filename"`, and generates the hash for `("username$filename$1"` ... `"username$filename$n")`. The download uses a **getMulti** function to concurrently retrieve multiple items from the Memcached server pool.

5.3.1 Integration of the overload control solution

To apply the overload control solution in the Distributed Fileserver, we deploy two agents within the nodes of the case study:

- **A capacity monitoring agent**: This component runs within the datastore nodes, and implements the *Distributed Capacity Monitoring* component of the solution (see Section 5.2.2).

- **An admission control agent**: This component runs within the application nodes and implements the *Distributed Admission Control Algorithm* (see Algorithm 5, Section 5.2.3) with a specialized *Resource Location Discovery* phase).

The *Overload Control Distributed Memory* can be implemented either by a new set of standalone nodes, or by the existing tiers. During the evaluation of the solution, we will use the first option (i.e., the distributed cache is managed by a standalone pool of Memcached nodes) in order to assess the overhead of this component (as further discussed in Section 5.5). As we discussed in Section 5.2, the *Data Location* block implementation depends upon the application type. For this case study (see Figure 5.11), the location algorithm parses the HTTP Request message, to extract information such as the request type, the Username, the Filename and the content-length. Then, with this information it builds the same group of hash-keys generated by the application (e.g., `user8`, `user8$file.txt`, `user8$file.txt$chunk1`, ...).

Using the request identifier (i.e., `username` for the register and the unregister operations, and `username$filename` for the upload and the download operations) the agent performs a lookup to retrieve the location of all the resources required to complete the user request. In case of cache miss, the agent finds these location by applying the hash function (`MD5`) to all the hash-keys, and creates a new entry in the Data Location Cache. Once the locations have been computed, it returns to the Admission Control Algorithm an array with the number of storage requests that are going to be performed on each storage node (denoted with L in the previous section). In this context, the $i-th$ position of the array represents the number of file blocks to be accessed on the storage node i.

5.3.2 Experimental evaluation

The experimental testbed infrastructure consists of eight host machines SUPERMICRO (high density), equipped with two 8-Core 3.3Ghz Intel XEON CPUs (32 logic cores in total), 128GB RAM, two 500GB SATA HDD, four 1-Gbps Intel Ethernet NICs, and a NetApp Network Storage Array equipped with 32TB of storage space and 4GB of storage SSD cache. The hosts are connected to three 1-Gbps Ethernet network switches, designed for management, storage and VM network traffic respectively. The infrastructure is managed by OpenStack Mitaka and the hosts are equipped with VMware ESXi 6.0 hypervisor.

In order to reproduce representative unbalanced overload conditions, we

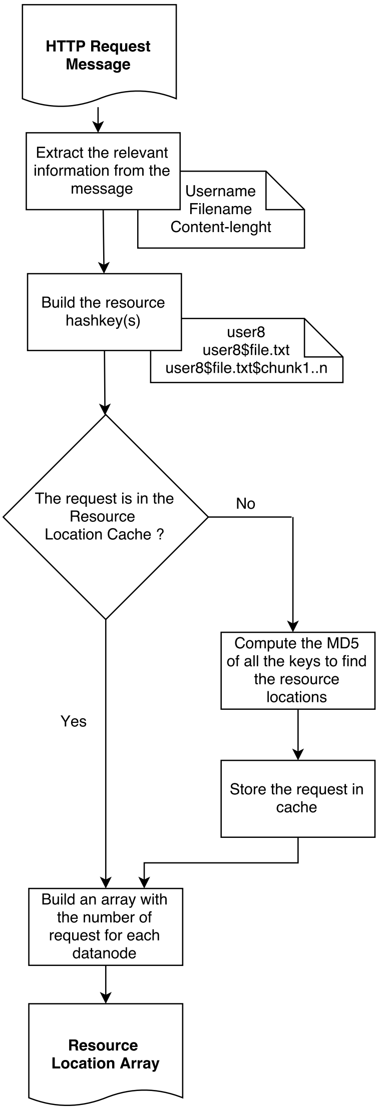

HTTP Request Message
Extract the relevant information from the message
Username
Filename
Content-lenght
Build the resource hashkey(s)
user8
user8$file.txt
user8$file.txt$chunk1..n
The request is in the Resource Location Cache ?
No
Yes
Compute the MD5 of all the keys to find the resource locations
Store the request in cache
Build an array with the number of request for each datanode
Resource Location Array

Figure 5.11. Resource Location Discovery logic for the distributed fileserver case study

setup a testbed with a large number of nodes. In detail, we scaled both the application and the datastore tiers up to 50 independent VMs. The application

Table 5.1. Configuration of the experimental Fileserver testbed

Node Type	# of nodes	VM configuration
HAProxy (frontend)	10	1 vCPU, 4GB RAM, 20 GB HDD
Fileserver (application)	50	1 vCPU, 4 GB RAM, 20 GB HDD
Memcached (datastore)	50	1 vCPU, 2 GB RAM, 20 GB HDD
JMeter runner (workload generator)	10	4 vCPU, 8 GB RAM, 40 GB HDD
JMeter master (workload controller)	1	4 vCPU, 16 GB RAM, 40 GB HDD,
Total	*140*	*154 vCPU, 436 GB RAM, 2.6 TB HDD*

requests are balanced to the application nodes by a front-end tier composed by 10 VMs running the HAProxy load balancer. The details of the resources allocated to each VM is given in Table 5.1, in which we indicate the number of node replicas in each tier, and the resources of the VM (virtual CPU, memory and storage).

The workload is generated by Apache JMeter through a distributed setup. To this purpose, we deployed ten additional VMs that submit requests to the system, and a controller VM to set-up the experiment and collect performance data, such as application latency, throughput and service failures. Each JMeter VM submits requests on a specific load balancer instance, at frontend tier. The requests are then balanced in round-robin fashion to the nodes of the application tier. The JMeter scenario reproduces a set of clients that initially register on the system, and that then perform a session of uploads and downloads of random files. The size of a file is randomly distributed between 8Kb and 4Mb. With the above configuration, the system can handle up to 700 concurrent users with no failures, corresponding to an average throughput of 175 uploads/s and 175 downloads/s.

Each experiment lasts 9 minutes and it is divided in three phases, as exemplified in Figure 5.12:

1. **Initial ramp-up phase** (2 min): In this phase, we gradually introduce new clients in the system, up to the engineered capacity, and we wait for a steady state. We then use this condition as starting point for the evaluation.

2. **Hot-spot generation** (5 min): In this phase, we vary the workload by introducing groups of users accessing shared file resources, with the purpose of generating an hot-spot in the storage tier.

3. **Final ramp-down phase** (2 min): In this phase, we gradually reduce the unbalanced workload until we remove the effects of phase 2.

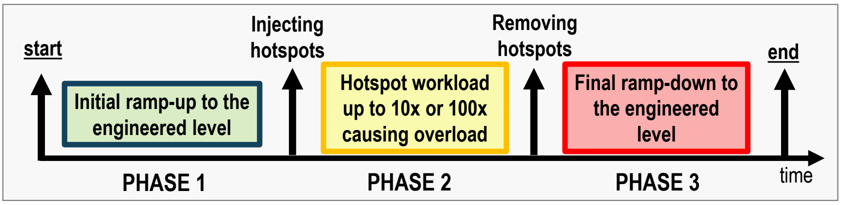
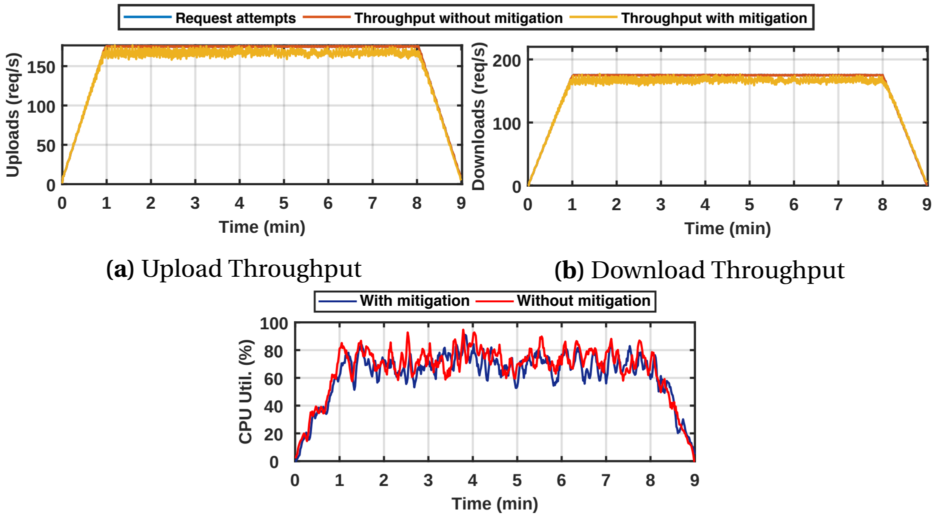

start
Initial ramp-up to the engineered level
Injecting hotspots
Hotspot workload up to 10x or 100x causing overload
Removing hotspots
Final ramp-down to the engineered level
end
time
PHASE 1
PHASE 2
PHASE 3

Figure 5.12. Phases of the evaluation experiments.

To evaluate the performance of the proposed overload control solution, we designed an experiment plan with unbalanced overload conditions in the storage tier caused by the client workload (hot-spots). Initially, we apply a workload with a balanced request mix at a rate within the engineered capacity, in which each user requests its own random files; then, we apply a skewed workload, in which groups of users repeatedly access a shared set of files, at a rate 4, 10 and 100 times the engineered capacity. We reproduced the same scenarios with and without our solution, by varying the number of hot-spot clients injected during the phase 2 between 0 (i.e., balanced workload) and 70K (100x skewed workload). We performed in total 8 experiments, lasting 9 min each. We first present the balanced case at the engineered capacity; then, we discuss the effect of the hot-spot injection, with reference to a representative case at 10x the engineered capacity; finally, we compare the overall throughput across all the experiments, with and without our solution enabled.

The experiment with the workload at the engineered level (1x) confirms that our solution works as expected under normal conditions (Figure 5.13a), since the upload throughput of the system is still at the engineered capacity, and no failures are experienced. Similarly, the throughput of the downloads at steady state is the same of uploads (Figure 5.13b), with no failures.



(a) Upload Throughput

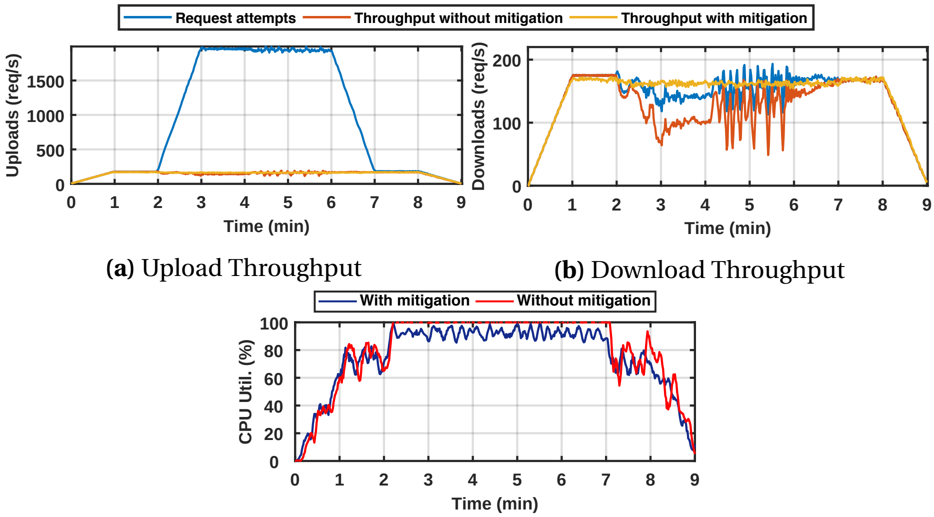
(b) Download Throughput

(c) Memcached CPU consumption

Figure 5.13. Fileserver performance at the engineered capacity (1x)

To reproduce the hotspot scenario, we consider a workload that exceeds the engineered capacity, by applying workload surges about 10x the engineered level. To this purpose we add a new group of users accessing shared files, causing hot-spots on 7 out of 50 datastore nodes. Without the overload control solution, both the upload (Figure 5.14a) and download (Figure 5.14b) throughput degrade significantly. Since the two operations are not independent, the download throughput is degraded because of both the growth of the latency and failures of upload operations. At the 10x overload, the hotspot users (min 4-7) also affect normal users. During the initial peak, the system tries to process all hot-spot requests, but failing at completing most of them. This behavior is also visible by looking at the CPU utilization of a hotspot node during a 10x overload (Figure 5.14c). In the case of no overload control, the CPU utilization pathologically saturates to 100%; instead, with overload control, the CPU utilization stabilizes at 85%, which is the target CPU utilization that we configured in the *Admission Control Agent* to avoid excessive resource competition. Finally, after the hot-spot phase, the CPU utilization reduces as expected in both cases (min 7-9). Our solution ensures that the performance matches the engineered level both for the upload throughput and download throughput. By preventing the admission of the requests that are going to require hot-spot datastore nodes, the overload control solution leaves the system with enough available capacity to serve all the other requests, even during a workload surge of ten times the capacity. We found that the system exhibits the same behavior under the other levels of skewed workload surges (i.e., from 4x to 100x).

In the Figure 5.15, we summarize all of the results obtained with the Fileserver case-study. When the overload mitigation is not deployed (*without mitigation*), the fileserver exhibits a noticeable throughput degradation. In the worst case (100x overload), the throughput reduces to about one half on average for uploads, and to about one third on average for downloads. Instead, with the overload mitigation enabled, the throughput is always close (i.e., it is higher than 90%) to the engineered throughput, even in the worst case of the



(a) Upload Throughput

(b) Download Throughput

(c) Memcached CPU consumption

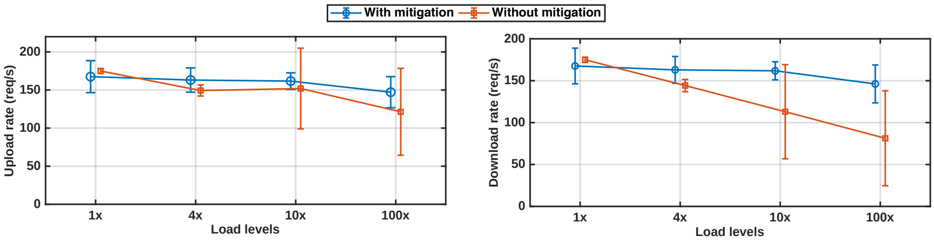
Figure 5.14. Fileserver performance at 10x engineered capacity

100x overload condition.

5.4 The IP Multimedia Subsystem Case Study

The IP Multimedia Subsystem (IMS) is an architectural framework for delivering multimedia services over internet. The current vision of telecom operators, and the focus of research, is to adopt cloud computing technologies in the telecommunication industry [125]. As a matter of fact, many telco operators are migrating and upgrading their systems to benefit from the emerging cloud paradigm. The key enabling technology for the cloud is **virtualization**: the telecom operators are thus attempting to virtualize the IMS infrastructure to optimize costs, performance and management of services and equipments.

In this section, we will analyze the impact of unbalanced overload conditions in the context of the *Clearwater* project [46]. We deployed the proposed



(a) Upload Throughput

(b) Download Throughput

Figure 5.15. Summary of the results on the distributed fileserver, with and without the overload control solution

solution on top of the Clearwater IMS. In particular, we consider the communication between Sprout (S-CSCF) and Memcached nodes since reflects the view of a modern multi-tier architecture. The IMS users (*subscribers*) access its own data (e.g., authentication and billing information), and user requests are balanced across the datastore tier through consistent hashing. Thus, there are no hot-spot resources in this case study; instead, unbalanced overloads can be caused by resource hogs and by configuration issues.

Since Clearwater is an implementation of IMS, its architecture (showed in Figure 5.16) reflects the traditional IMS architecture, but with notable differences. In particular, all components are horizontally scalable using simple, stateless load-balancing; most long-lived state is stored in back-end nodes using storage technologies such as Cassandra [126]; interfaces between the various components use connection pooling.

The Clearwater IMS includes a throttling mechanism, which rejects requests in excess to avoid overloading a node [127, 128]. It uses a *token bucket* to control the rate of requests that a node is allowed to process. The token replacement rate is tuned by measuring the latency for processing requests, and by comparing, every twenty requests, this measure with a configured latency target. Clearwater adopts a variation of the algorithm proposed by Welsh and Culler [31], by using a smoothed mean latency to compare with the latency tar-

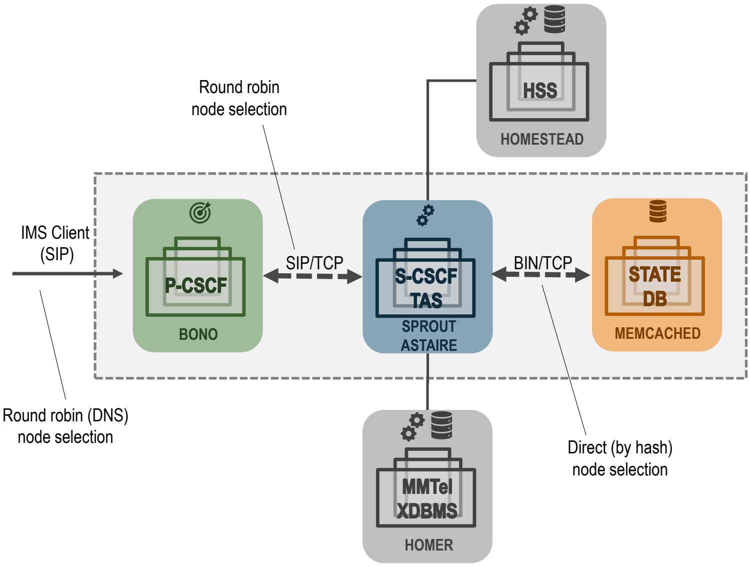


Figure 5.16. Clearwater IMS Components

get. In our analysis, the experiments labeled as "*without mitigation*" represent a standard Clearwater installation inclusive of this overload control mechanism, which we compare to our resource-aware overload control solution.

5.4.1 Integration of the overload control solution

To apply the overload control solution, we deploy two agents:

- **A capacity monitoring agent**: This component runs within the Memcached nodes and implements the *Distributed Capacity Monitoring* component of the solution.

- **An admission control agent**: This component runs within the SPROUT nodes and implements both the Distributed Admission Control Algorithm (see Algorithm 5 with a *Resource Location Discovery* phase specialized for the IMS. This agent can also run on BONO VMs (i.e., along with the P-CSCF function).

The *Overload Control Distributed Memory* has been implemented by a new set of standalone nodes (i.e., a standalone pool of Memcached nodes). This separation allows us to assess the overhead of this component (Section 5.5).

The Resource Location Discovery block implements the procedure described in the flowchart of Figure 5.17. It extracts the user identity (e.g., `50012345@example.com`) and, in case of an INVITE message, the identity of the callee (e.g., `5001244@example.com`) from the incoming SIP message.

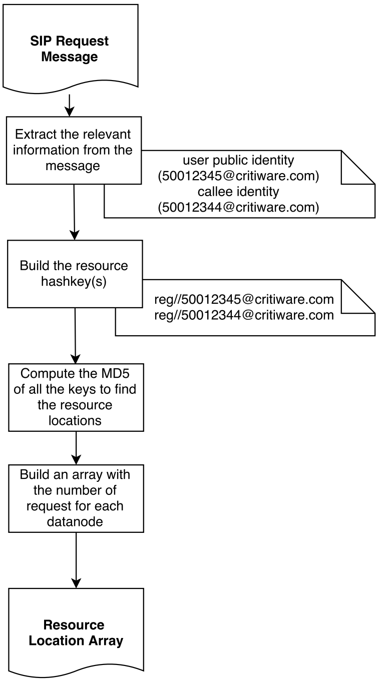

SIP Request Message
Extract the relevant information from the message
user public identity (50012345@critiware.com) callee identity (50012344@critiware.com)
Build the resource hashkey(s)
reg//50012345@critiware.com
reg//50012344@critiware.com
Compute the MD5 of all the keys to find the resource locations
Build an array with the number of request for each datanode
Resource Location Array

Figure 5.17. Resource Location Discovery logic for the IMS case study

On each user request, the Sprout node accesses the information about the user session in JSON format, by performing a single query on Memcached, using the key reg\\user_identity (e.g., reg\\50012345@example.com).

The *Resource Location Discovery* phase finds the right Memcached node by applying the hash function to the key (e.g., MD5(reg\\50012345@example.com)). The Data Location Cache here is optional, as the hash function can be computed on every request with a small overhead. The Distributed Admission Control Algorithm takes into account the type of the request and rejects only the first REGISTER and the first INVITE messages, in order to give priority to the already established SIP sessions. Since the data location can be determined solely from the request, we use the Distributed Memoory component to store and update only the capacity information about the datastore nodes.

After an REGISTER or an INVITE message, the IMS and the user agent generate a flow of messages that is pre-defined by the SIP protocol. We can rely on the fact that the SIP protocol generates the same flow of messages (e.g., INVITE - 100 Trying - 180 Ringing - 200 OK - ACK). In all these messages, the server will request the same key and thus the same node. Thus, in order to accept a first INVITE message, we check to have enough capacity to satisfy all the subsequent messages. We apply the admission control at the first message of a SIP session, in order to avoid user-perceived errors in the middle of a SIP session.

5.4.2 Experimental evaluation

The experimental testbed consists of the same hardware and software of the previous case study (Section 5.3). In order to reproduce unbalanced overload conditions, we defined a large IMS installation, which includes 50 nodes in the Sprout application tier, 50 nodes in the Memcached storage tier, and 10 nodes in the Bono front-end tier. We configured the number of nodes for the other components of Clearwater proportionally to the capacity of the Sprout, Bono, and Memcached nodes, such that to have an average CPU utilization in each component of 80% and no request failures. Moreover, we deployed a

Table 5.2. Configuration of the experimental IMS testbed

Node Type (component)	# of nodes	VM configuration
Bono (P-CSCF)	10	1 vCPU, 4GB RAM, 20 GB HDD
Sprout (S-CSCF + astaire)	50	1 vCPU, 2 GB RAM, 20 GB HDD
Memcached (DB)	50	1 vCPU, 2 GB RAM, 20 GB HDD
Homer (MMtel XDBMS)	10	1 vCPU, 8 GB RAM, 40 GB HDD
Homestead (HSS)	10	4 vCPU, 8 GB RAM, 40 GB HDD,
SIPp (workload gen.)	10	1 vCPU, 2 GB RAM, 20 GB HDD
Total	*140*	*170 vCPU, 420 GB RAM, 2.2 TB HDD*

cluster of 10 SIPp workload generators to generate the IMS workload to the system. The workload reproduces the typical message flows between subscribers, according to the SIP protocol. These flows are also adopted to test the Clearwater IMS, and the complete scenario used in our tests is available online [118]. The details of the resources allocated to each VM are given in Table 5.2, in which we indicate the number of node replicas (one VM for each replica) in each tier, and the resources of the VM (vCPU, memory, storage).

Each SIPp instance generates SIP traffic towards a specific P-CSCF instance. The IMS scenario reproduced with the workload generator is the following: every subscriber registers and periodically renews the registration every minute, on average. After a successful registration, a subscriber attempts to set up a call with another subscriber (with 16% of probability) or remains idle until the next registration renewal (with 84% of probability). The call hold time is, by default, 60 seconds. Thus, the scenario reproduces 60 Busy Hour Register Attempts (BHRA) per user and 5 Busy Hour Call Attempts (BHCA) per user. Then, we vary the number of subscribers to soliciting the system with different levels of load. As this work has been conducted in the context of a R&D cooperation with an industrial organization (a major vendor of cloud and NFV products and services), we tuned the parameters of this workload (e.g., the rate of busy-hour call attempts) according to the experience of our industrial partner with overload conditions [80].

With the above workload configuration, our deployment can handle up to 110k subscribers without exhibiting any failure, corresponding to 1,833 REGISTER/s and 153 INVITE/s on average, with an average CPU consumption (measured in the application nodes) of 80%. In the following, we refer to this load level as engineered capacity.

Each experiment lasts 16 minutes and it is divided in three phases, as exemplified in Figure 5.18:

1. **Initial ramp-up phase** (4 min): In this phase, we gradually introduce new subscribers in the system, up to the engineered capacity and we wait for a steady state. We then use this condition as a starting point for the evaluation.

2. **Workload surge** (6 min): In this phase, we vary the number of subscribers, e.g. to reproduce a workload surge (up to 10 or 100 times the engineered capacity) causing the overload of the application tier.

3. **Hog injection** (6 min): In this phase, we inject in a subset of storage nodes a busy wait in the Memcached request handling code to simulate unbalanced overload conditions, such as reduced capacity due to background tasks (*hogs*) or an incorrect capacity planning or configuration of a subset of machines.

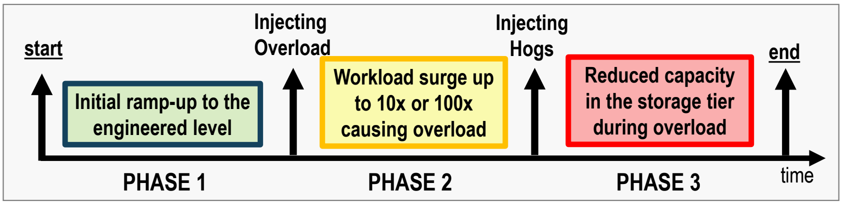

start
Initial ramp-up to the engineered level
Injecting Overload
Workload surge up to 10x or 100x causing overload
Injecting Hogs
Reduced capacity in the storage tier during overload
end
PHASE 1
PHASE 2
PHASE 3
time

Figure 5.18. Phases of the evaluation experiments.

To evaluate the performance of the overload control solution we designed an experiment plan to evaluate the following scenarios:

- During a normal workload (at 70% and 100% the engineered capacity), we emulate reduced capacity in 5 out of 50 storage nodes causing unbalanced load. We evaluate the ability of the solution to prevent request failures in the storage tier due to the CPU saturation of the slower Memcached nodes, ensuring the success of user session requiring the faster storage nodes.

- During a workload surge (at 4, 10 and 100 times the engineered capacity), we emulate reduced capacity in 5 out of 50 storage nodes causing unbalanced load. We evaluate the ability of the solution to protect application nodes from the workload surge by discarding the excess of the requests and, at the same time, ensuring the engineered throughput despite the load unbalance in the storage tier.

We reproduced these scenarios with and without our solution enabled by varying the number of subscribers in phase 2 of the experiment according to the above 5 levels, ranging from 0.7x (i.e., 80k subscribers) to 100x (i.e., 11M subscribers). Thus, we performed in total 10 experiments, lasting 16 min each.

We first present the details of a representative experiment of the first group (at 1x load), discussing the effect of the unbalanced load on the storage; then, we discuss the combined effect of a workload surge, by presenting the detail of a representative experiment of the second group (at 10x load); then, we compare the overall IMS throughput through all the experiments, with and without our solution enabled.

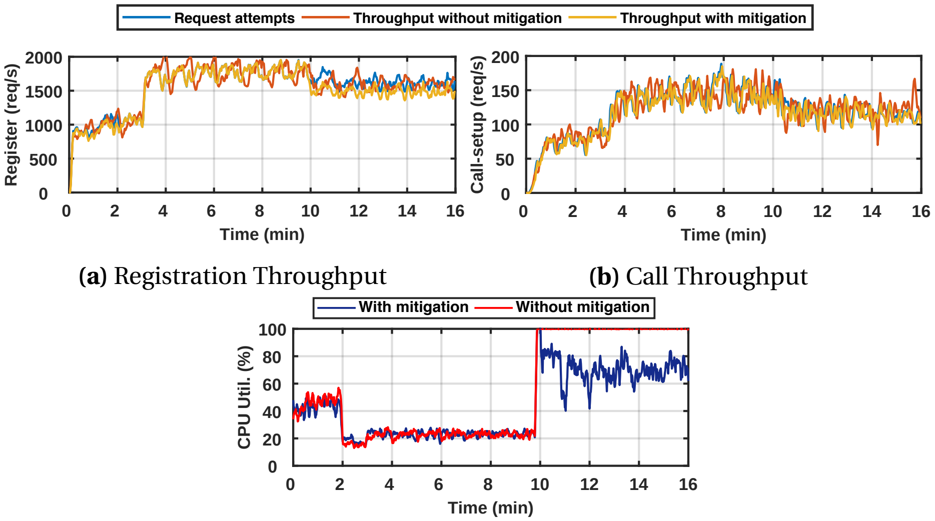
Figures 5.19a and 5.19b show the performance of the IMS system for registration requests, with a workload at the engineered level (i.e., 1x), by generating 110k subscribers during the first 2 minutes.

During the steady state of phase 2 (starting at minute 4), after all the subscribers performed an initial registration, the system is able to process 1800 registrations/s and 150 call-setup/s on average. The throughput with the overload control solution matches the throughput without the solution: thus, the solution does not interfere with the system under normal conditions.

When the hog is enabled (at minute 10), the registration throughput decreases by 12% (both in the cases with and without mitigation). Our solution detects and discards these requests before they enter in the system, avoiding the overload of the storage tier. Instead, without our solution, most of the requests experience failures due to the overloaded storage nodes. This situation should be avoided since it might cause consistency issues across the storage tier (i.e., some storage nodes are updated while other storage nodes cannot be updated). Thus, the overload control solution can prevent these consistency issues. The effect of overload mitigation is more evident in Figure 5.19c, showing the CPU consumption with and without the overload control enabled of one of the five Memcached nodes slowed down during the experiment (i.e., with a CPU hog injected in the memcached code). The traffic throttling performed in the application tier prevents the saturation of the CPU in the Memcached instances with the CPU hog enabled, and stabilizes the average CPU utilization at 75%.

In Figures 5.20a and 5.20b, we show the performance of the IMS under a workload that exceeds the engineered capacity. After the first phase, at minute 4, we generate the workload surge by about 10 times the engineered level, by provisioning up to 1.1M subscribers. Without mitigation, the registration throughput is very low (about 20 registrations per second), while none of the registered users is able to make a call. With the mitigation enabled, the throughput is always above the engineered throughput, for both registration and call-setup operations.

It is worth noting that during the intermediate phase of the experiment



(a) Registration Throughput

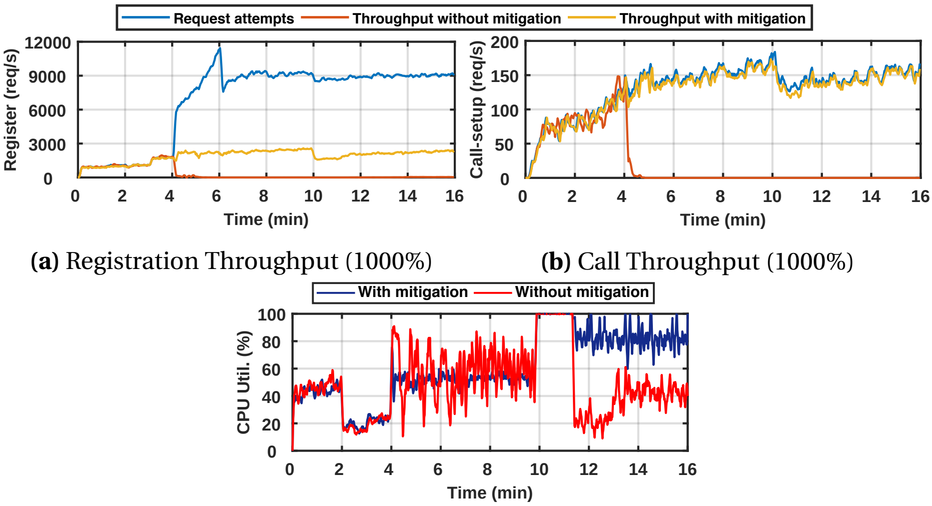
(b) Call Throughput

(c) Memcached CPU consumption

Figure 5.19. Timeseries IMS at the engeenered capacity

(workload surge), the registration throughput is even higher than the engineered level (close to 2000 regs/s). In this phase, most of the new registration requests are discarded. However, re-registration requests of the previous sessions require fewer datastore requests than new registrations (a new registration requires 6 database accesses, a renewal requires only two accesses). Since the load on the datastore is lower, our solution can accept some new users. With the hog injected, our solution quickly discards requests that require the slower nodes, and it is able to preserve the registration throughput for the already registered requests (Figure 5.20a).

Even in the last phase of the experiment (during the hog injection) the throughput is still above the engineered level. The reason is that each session makes some requests to several storage nodes, and if any of these storage nodes is overloaded, the entire session is not admitted. In this way, we are able to prevent the requests to the overloaded storage nodes, and also to prevent some of the requests to the non-overloaded storage nodes. For this reason, it could be possible that some storage nodes become less loaded, so other sessions (i.e., the ones that do not use the overloaded nodes) are gradually admitted in place of the rejected ones. The CPU utilization of Memcached node (shown in Figures 5.20c) is stable at 50% during the phase 2. At beginning of phase 3, the hog slows down also all the requests currently being served by the node. This causes the saturation of the CPU for 1 minute, both with and without the solution. During this time, with our solution, the admission control does not accept any new request that requires resources on the overloaded Memcached nodes. Thus, when this effect ends, part of the requests are admitted to the system and the CPU utilization becomes stable at 80%. Vice versa, without our solution, the application nodes keep submitting new requests to the storage nodes causing the exhaustion of the socket pool, since all the requests are waiting for a response. This, in turn, causes some application nodes to fail.



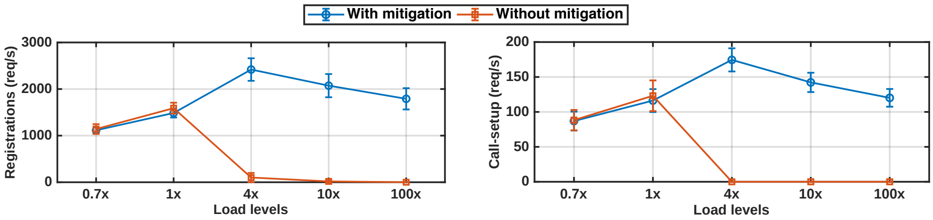
(a) Registration Throughput (1000%)

(b) Call Throughput (1000%)

(c) Memcached CPU consumption

Figure 5.20. Timeseries IMS at 10x the engeenered capacity

In the Figure 5.21 we summarize the performance of the IMS, for all the considered load levels, both with and without the overload control solution during the phase 3. The Figures 5.21a and 5.21b provide the average throughput for the IMS registrations and the IMS call-setups, respectively. The error bars indicate the standard deviation of the throughput. Without workload surges (i.e., levels 0.7x-1x) there are no significant differences between the experiments with and without mitigation. However, as discussed before for the 1x case, without mitigation the excess of the requests exhibit failures at the storage tier, while with mitigation enabled, those requests are discarded by the admission control, before they enter in the system, thus preventing the overload of the storage tier nodes. Under high load conditions (i.e., 4x, 10x, 100x load levels), the Clearwater IMS components experience software crashes due to resource exhaustion (especially in the frontend Bono nodes). When we enable the overload control solution, the IMS does not experience any crash, and it can reach a stable throughput above the 90% of the engineered capacity.



(a) Registration Throughput

(b) Call Throughput

Figure 5.21. Summary of the results with and without the overload control solution, during unbalanced datastore overload

5.5 Overhead and scalability of the overload control solution

The solution requires an agent deployed on each VM of the application tier to perform a fine-grained admission control. This agent has a small memory footprint (less than 10 MB during the 100x overload case). The request inspection and admission requires some CPU resources, depending on the volume of the incoming load; in our experiments, the CPU overhead was always small (less than 8% during the 100x overload case).

In the overload control solution, we used a **distributed memory** as a shared memory among all the nodes, to store the state of the overload control algorithm and the state of the nodes. Since the solution is meant to be deployed in systems with a big number of nodes in each cluster (with 10k nodes in production systems of our industrial partners), we designed this component to achieve a high scalability and low overhead.

The complexity of the control algorithm does not depend on the number of application nodes, nor storage nodes. Indeed, the algorithm finds the location of the required nodes by extracting the metadata from the user request message. Then it retrieves the current available capacity of these nodes to check if there is enough available capacity in all of them. If the request can be accepted, then it updates the capacity. Therefore, the number of operations does not increase by scaling up the storage tier.

In detail, for the **Clearwater** case study, the solution requires to query (get) the capacity of:

- 4 storage nodes for an initial registration
- 2 storage nodes for a registration renewal
- 1 storage node for a call setup

In Figure 5.22 we show the average CPU consumption of the control agent component across all experiments in the IMS case study.

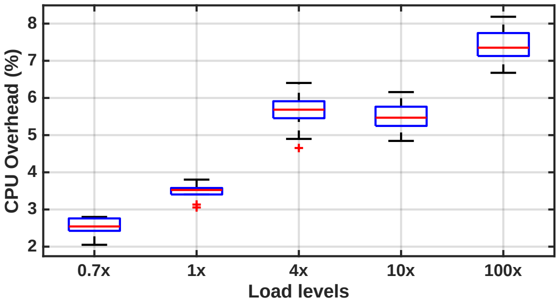


Figure 5.22. Overhead of the mitigation agent at each load level for the IMS Case Study

For the **Distributed Fileserver** case study, the number of accesses depends on the number of blocks in which the file is divided, but is the same for an upload and a download request. According to our test configuration, the average number of blocks per file is 8. In detail:

- 1 set request to update the *Distributed Memory* and B requests according to the locations of the file blocks in case of a upload request.
- 1 get request to check the *Distributed Memory* and B requests according to the locations of the file blocks in case of a download request.

In Figure 5.23 we show the average CPU consumption of the control agent component during all of the analyzed cases, for the Distributed Fileserver scenarios.

To achieve scalability, the Distributed Memory creates a fixed pool of persistent connections per-node; thus, the number of connections to the Distributed Memory only grows linearly with the amount of nodes in the cluster. We avoid any direct communication between pairs of nodes in the tiers.

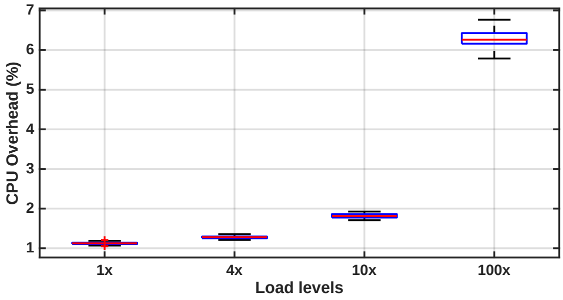


Figure 5.23. Overhead of the mitigation agent at each hotspot scenario for the Fileserver Case Study

Moreover, the Distributed Memory is based on a separate Memcached datastore. In this way, the Distributed Memory can benefit from the scalability features of Memcached. In particular, Memcached can distribute the datastore of the Distributed Memory across several nodes, thus increasing the capacity and avoiding that the Distributed Memory becomes a performance bottleneck. Moreover, Memcached can transparently manage the distribution of key-value pairs, and can optimize the memory consumption of the datastore.

5.5.1 Sizing the Distributed Memory

In this Section we derive analytic relationships to describe the expected load on the Distributed Memory component at different scales. The objective of this analysis is to obtain a practical formula to quantify the number of nodes to deploy, in the worst case.

Let it T_A, the **engineered throughput of a single application node**. Let it N_A and N_C the **number of application and distributed memory nodes**, respectively. Let it $\overline{R}$ the **average number of accesses per service request**.

Each application node maintains a pool of p connections to each of the N_C storage nodes (this avoid opening TCP connections on-demand, since it is a costly operation). Therefore, the total number P of connections to each node is constant ($P = N_A * p$).

the maximum number of requests performed to the Distributed Memory is limited by the engineered capacity of the application tier. If requests exceed the capacity of the application node, they are rejected without any further inspection. The maximum number of requests inspected by an agent is equal to the engineered capacity of the application node.

We denote the **maximum throughput of a Distributed Memory node** at concurrency level P as $T_C^{(P)}$. This throughput can be obtained by performing a simple experiment with a synthetic workload generator (such as the *memtier-benchmark* tool), by reproducing a workload with P concurrent connections and measuring the corresponding throughput (see Section 5.5.2 for the experimental results).

The maximum number of requests that the whole Distributed Memory can handle is $N_C * T_C^{(P)}$. The average number of requests performed by the application tier at the engineered level is $N_A * T_A * \overline{R}$. To ensure a correct sizing of the system, the number of requests that the Distributed Memory should be handle needs to be at least equal to the average number of requests performed by the whole application tier (at the engineered level):

$$N_A T_A \overline{R} = N_C T_C^{(P)}$$

Therefore, we can find the **minimum number of required Distributed Memory nodes** as:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil$$

5.5.2 Example: scaling the solution up to 10K nodes

We performed a simulation in order to estimate the maximum throughput of a Distributed Memory node at different contention level. We used the open-source *memtier-benchmark* tool to generate a Memcached synthetic workload. We configured the workload to be an equal mix of “set” and “get” requests, since the admission control solution performs updates (a “get” followed by a “set” on the same key).

Figure 5.24 shows the values of the throughput $T_C^{(P)}$ for increasing values of the number of concurrent connection P, ranging from 50 to 50k.

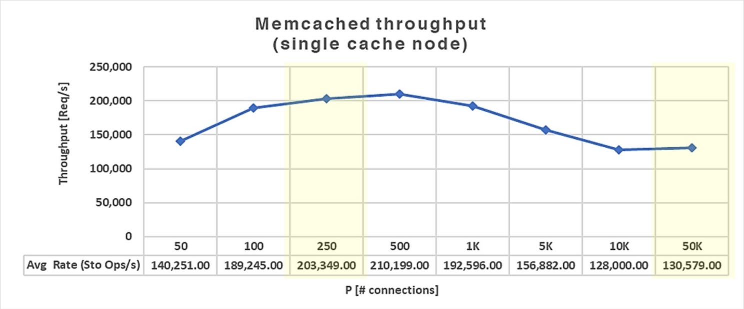


Figure 5.24. Performance of a Distributed Memory server at increasing level of concurrent connections (up to 50K)

In the following we show how to use the model described in Section 5.5.1 and the simulation results, to figure out the minimum number of datastore nodes required to scale to 10k nodes.

A) Our testbed configuration (up to 50 nodes). The number of application nodes is $N_A = 50$. The average number of requests to the Distributed Memory per service request is $\overline{R} = 4$. The engineered throughput of a single application node is $T_A = 40$ req/s at the steady state (the whole throughput is 2,000 req/s). The connection pool size in each application node is $p = 5$ connections. The concurrency level at Memcached server is $P = N_A p = 250$ connections. Each server is configured with 1 vCPU and 4GB RAM. The performance of a single Memcached server at this concurrency level is $T_C^{(250)} = 203,349$ req/s.

Therefore, the minimum number of nodes to deploy is:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil = \left\lceil \frac{50 * 40 * 4}{203,349} \right\rceil = \left\lceil \frac{8,000}{203,349} \right\rceil = 1$$

B) Scaling the system up to 10k agents. Under the same conditions, if we scale the application tier up to $10k$ nodes ($N_A = 10,000$), we obtain a Memcached concurrency level $P = N_A p = 50,000$ connections. Considering that a single TCP connection requires up to 1 kb of RAM in a Linux system, the memory overhead will be less than 50 MB per node. However, the performance of a Memcached server under this level of contention will drop to $T_C^{(50K)} = 130,579$ req/s. Therefore, the minimum number of Memcached nodes that are required to handle 10,000 agents will be:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil = \left\lceil \frac{10,000 * 40 * 4}{130,579} \right\rceil = \left\lceil \frac{1,600,000}{130,579} \right\rceil = 13$$

This is a **worst-case** result. On average a big part of the 50,000 connections are idle most of the time: the concurrency level is lower than the number of active connection, since the connection pools can have spare connections.

Figure 5.25 shows the required number of nodes for increasing number of application nodes (N_A) and for different values of $\overline{R}$, assuming that each application nodes has the same engineered throughput as the previous examples (i.e., $T_A = 40$ req/s). All the configurations below 200 application nodes can correctly work with a single Memcached node. In the extreme case with a deploy with 10,000 application nodes, in which a single application request accesses 20 different storage nodes on average, the solution requires 62 Memcached nodes, assuming the performance of our test machine, reported in Figure 5.24.

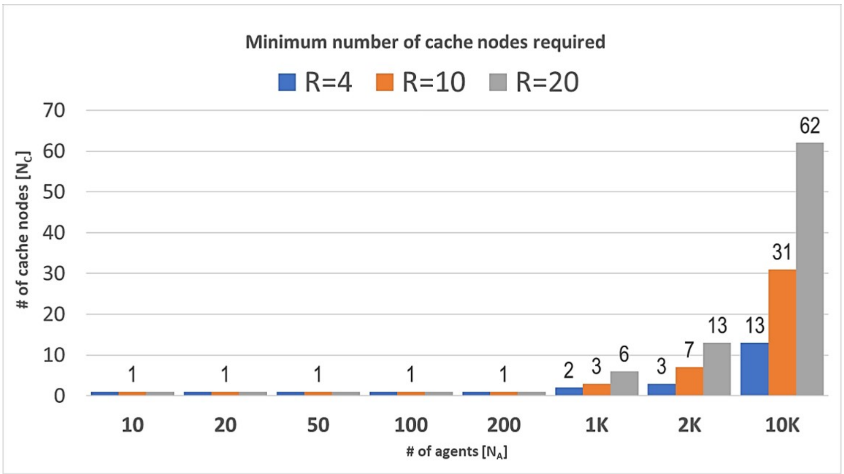


Figure 5.25. Minimum number of Distributed Memory nodes required by the solution, for different scale of application servers (N_A), by varying the average number of storage operations per each user request ($\overline{R}$)

5.5.3 Further optimizations

There are more strategies to improve the performance of Memcached at this extreme scale. These strategies can improve the throughput of each Memcached server, and at the same time reduce the level of resource contention.

1. **Use of a TCP concentrator proxy**, such as Twitter Twemproxy [129]. This middleware is a fast and lightweight proxy for Memcached and Redis protocols. It was built primarily to reduce the number of connections to the caching servers. It accepts requests coming from multiple connections and forwards them in pipeline on a smaller number of connections in order to achieve a lower concurrency and a higher throughput.

2. **Use of the UDP protocol**. Memcached can work on both TCP and UDP. The last is not very used, since it provides lower performance in case of a small number of concurrent connections. However, at the scale of 10K

agents, we expect a significant reduction of the system overhead and better performance than the TCP protocol.

The first strategy allows very a high throughput improvement, and it is widely used in production systems (e.g., by Twitter Twemproxy [129], Facebook [130] and Flickr [131]). However, it has the disadvantage that requires to deploy an additional component (the proxy). On the other side, the second strategy is simpler, it does not require additional resources and it is built in the Memcached code, but is not widely tested in production.

Chapter 6

Conclusion

A key challenge for telecom operators and service providers is to efficiently deploy rich network services, and to optimize network resources to improve customer's quality of experience. NFV solutions are expected to support extremely large scale architectures, providing high performance and high dependability. Overload management is a critical aspect of VNF systems that is affected by these concerns: if the system attempts to serve more traffic than its capacity, then each traffic unit will not be served with enough computing resources to meet Quality of Service (QoS) requirements, as specified in Service Level Agreements (SLAs); high-priority requests may experience failures; user sessions that were already admitted in the system may be disrupted, causing avalanche restarts and cascade failures due to retries and traffic handover; and the software becomes prone to failures due to resource unavailability, timeouts, and race conditions.

Therefore, to react timely to bottlenecks undermining the performance and the availability of the network services it is necessary optimizing the performance at very large scale without human intervention in response to both service configuration and workload variations and detecting the occurrences of network problems to mitigate their symptoms within few seconds. *The thesis has contributed to these issues with a threefold proposal.*

First, It presented an approach to ease the adoption of anomaly detection systems in production NFV services. I showed that, by taking into account the VNF service chain topology, it is possible correlating performance metrics from different VNFs to infer the health of the service chain (e.g., service performance anomalies caused by the occurrence of bottlenecks and component failures). I propose an algorithm to combine the correlations across multiple VNF replicas, to improve the accuracy of the detection. I validated the approach using an opensource NFV-oriented IP Multimedia Subsystem (IMS), namely Clearwater. I selected a set of scenarios including overload, contention on physical resources and crash of the VNF instances, and studied the impact on the quality of service. I evaluated the detection coverage (i.e., the percentage of the scenarios where the detection outcome is detected) and the detection latency (i.e., the time between the occurrence of a failure and the detection of the anomaly). The experimental results show that the approach performs well across several conditions when using the Running Moving Median (RMM) smoothing function and a window of 10 samples with a sampling period of 2s. With these parameters, an anomalous condition is detected within half minute on average, with a very high detection coverage and no false positives. The insensitivity of the algorithm against false positives, along with the freedom from thresholds that depend on the system (that would need to be calibrated with training samples, and to be tuned when the system is upgraded or reconfigured), are two key concerns that I took into account in the design of the algorithm, in order to make easier its adoption in production environments.

Second, the thesis proposed a novel framework, *NFV-Throttle*, for overload control in NFV services. This framework has been designed to support the service models of NFV (in particular, NVFIaaS and VNFaaS), by providing a set of *overload detection and mitigation agents* to be deployed either on VMs or on the physical hosts. These agents adopt simple and robust rules to control traffic drop and reject, by analyzing CPU utilization and the network traffic volume. Moreover, the proposed framework addresses not only workload peaks, but also physical resource contention, which can occur because of oversubscription of NFV services on the physical infrastructure, or because of faults that reduce the available resources. The contention is mitigated by tuning the priority of services that share the physical infrastructure. Moreover, I analyzed the problem of overload conditions caused by physical CPU contention within the guest VMs: this form of overload conditions have a different behavior (e.g., in terms of CPU utilization metrics) than the case of traffic spikes; and that the overload control solutions for traffic spikes can be ineffective, or even counterproductive, in the case of physical CPU contention. Therefore, I proposed an extention to the existing feedback control-based approach at guest-level to also address physical CPU contention to support VNFs deployed on IaaS, where the VNF has little visibility or control of the underlying infrastructure. This solution introduces a mechanism inside the VNF to occupy the CPU cycles freed by traffic throttling, in order to protect the feedback control loop from the opportunistic behavior of the hypervisor that may reclaim the CPU cycles. As done for the anomaly detection, I evaluated the proposed framework in the context of Clearwater. In the experiments, I considered stressful overload conditions with high workloads (up to 1000% of the nominal capacity of the system), and with resource hogs competing with the IMS for the physical resources. In all the scenarios, the proposed framework is able to achieve a high throughput, comparable to the maximum throughput under normal conditions, with a negligible memory and CPU overhead. Moreover, the overload control framework avoids failures of the NFV software that are triggered by stress and resource exhaustion. I also analyzed the relative benefits and the complementarity of VNF-level, host-level, and network-level overload control. The host-level control achieves the best performance, since it avoids the overhead of forwarding the traffic in excess to the VMs; however, the VNF-level control achieves comparable results, and can be applied in scenarios in which the physical infrastructure cannot be modified; finally, the network-level control allows to reject traffic at the boundaries of the NFV network, thus enabling the network to send notifications to clients and to neighbours about overload conditions, in order to gradually reduce the traffic in excess.

Last, the thesis analyzed the overload condition problems in modern large-scale architectures, especially when uniform load balancing among the nodes is not possible. In fact, during the interaction between stateless and stateful tiers typical problems are represented by hot-spot resources, unequal node configurations and unpredictable capacity variations due to background tasks or hogs. I propose DRACO, a distributed overload control solution for large multi-tier architectures, that dynamically monitors the capacity of the nodes to detect hot-spots and capacity variations that may be caused by resource hogs. Moreover, DRACO is designed to perform a fine-grained admission control by discovering the resources that will be required by a user request before its admission in the system. This will prevent unbalanced load conditions to cause overload in specific nodes when there is still available capacity in the tiers. The key innovative aspects of DRACO are:

- The solution is suitable to be applied to multi-tier systems, in which the traffic can only be filtered in the front-end tier (i.e., the application tier). In these multi-tier systems, an internal tier (in particular, the storage tier) should not drop any traffic; otherwise, it would cause inconsistencies between the application and storage tier, and among nodes in the storage tier. The solution has been designed to map the application requests to storage resources and to only drop requests at the application tier, without loss of consistency.

- The solution mitigates overload conditions that are unbalanced, that is, the overload only affects specific nodes in the system. The solution has been designed to make the most efficient use of the capacity of the storage tier, by admitting user requests that only use non-overloaded storage nodes.
- The solution can be applied without making any change to the storage tier (thus, it is suitable to be applied even if the system uses "off-the-shelf" key-value data store technologies).
- The solution uses simple and robust heuristics to estimate the number of requests that can be accepted by the multi-tier system. The heuristics are easy to deploy in production since they only require generic metrics for CPU utilization and network bandwidth utilization.

I evaluated DRACO by means of two case studies: a Distributed Fileserver, which is very sensitive to problems of data consistency and hot-spots, and the Clearwater IMS, which require carrier-grade levels of performance and availability. For both the case studies I performed two groups of experiments: The first group is with a load up to the capacity and an unbalanced condition among the nodes of the tiers, in order to evaluate the performance of the control system in absence of overload. The second group is with a load up to 100 times higher than the engineered capacity of the system, in order to evaluate the performance (and the overhead) of the control system during extreme overload conditions. Results show that the solution quickly reacts to bottleneck changes, and preserves more than the 90% of the throughput during the most severe overload conditions and prevents service failures due to resource exhaustion in the nodes.

Bibliography

[1] ETSI, NFVISG, "GS NFV-MAN 001 V1. 1.1 Network Function Virtualisation (NFV); Management and Orchestration," 2014.

[2] OpenStack Foundation Report, "Accelerating NFV Delivery with OpenStack," 2016. [Online]. Available: https://www.openstack.org/assets/telecoms-and-nfv/OpenStack-Foundation-NFV-Report.pdf

[3] E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*, 1st ed. Wiley-IEEE Press, 2012.

[4] D. Cotroneo, L. De Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, "Network function virtualization: Challenges and directions for reliability assurance," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on.* IEEE, 2014, pp. 37–42.

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

[6] T. Niwa, M. Miyazawa, M. Hayashi, and R. Stadler, "Universal fault detection for NFV using SOM-based clustering," in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific.* IEEE, 2015, pp. 315–320.

[7] C. Sauvanaud, K. Lazri, M. Kaaniche, and K. Kanoun, "Anomaly Detection and Root Cause Localization in Virtual Network Functions," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th Conference on.* IEEE, 2016, pp. 196–206.

[8] L. Wang, R. A. Hosn, and C. Tang, "Remediating Overload in Over-Subscribed Computing Environments," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 860–867.

[9] S. A. Baset, L. Wang, and C. Tang, "Towards an Understanding of Oversubscription in Cloud," in *Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, 2nd USENIX Workshop on*, 2012.

[10] S. Singh and I. Chana, "Qos-aware autonomic resource management in cloud computing: a systematic review," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 42, 2016.

[11] ETSI, NFVISG, "ETSI GS NFV-REL 001 V1. 1.1: Network Functions Virtualisation(NFV); Resiliency Requirements," 2015.

[12] Doctor: Fault management and maintenance. [Online]. Available: http://artifacts.opnfv.org/doctor/docs/development_requirements/

[13] ETSI Industry Specification Group, "NFV Use Cases," 2014.

[14] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, pp. 10:1–10:42, 2010. [Online]. Available: http://doi.acm.org/10.1145/1670679.1670680

[15] S. M. Bridges and R. B. Vaughn, "Fuzzy data mining and genetic algorithms applied to intrusion detection," in *Proceedings of 12th Annual Canadian Information Technology Security Symposium*, 2000, pp. 109–122.

[16] A. Bivens, C. Palagiri, R. Smith, B. Szymanski, M. Embrechts *et al.*, "Network-based intrusion detection using neural networks," *Intelligent Engineering Systems through Artificial Neural Networks*, vol. 12, no. 1, pp. 579–584, 2002.

[17] M. Ramadas, S. Ostermann, and B. Tjaden, "Detecting anomalous network traffic with self-organizing maps," in *International Workshop on Recent Advances in Intrusion Detection.* Springer, 2003, pp. 36–54.

[18] M. Miyazawa, M. Hayashi, and R. Stadler, "vnmf: Distributed fault detection using clustering approach for network function virtualization," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on.* IEEE, 2015, pp. 640–645.

[19] B. R. Granby, B. Askwith, and A. K. Marnerides, "SDN-PANDA: Software-Defined Network Platform for ANomaly Detection Applications," in *Network Protocols (ICNP), 2015 IEEE 23rd International Conference on*, Nov 2015, pp. 463–466.

[20] C. Sauvanaud, K. Lazri, M. Kaâniche, and K. Kanoun, "Anomaly detection and root cause localization in virtual network functions," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, Oct 2016, pp. 196–206.

[21] ETSI Industry Specification Group, "NFV Resiliency Requirements," 2015.

[22] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum), “TL 9000 Quality Management System Measurements Handbook 4.5,” Tech. Rep., 2010.

[23] G. Galante and L. C. E. de Bona, “A Survey on Cloud Computing Elasticity,” in *Utility and Cloud Computing (UCC), 2012 IEEE 5th International Conference on*, 2012, pp. 263–270.

[24] P. C. Brebner, “Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications,” in *Performance Engineering, 3rd ACM/SPEC International Conference on*, 2012, pp. 263–266.

[25] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, “Scheduling strategies for optimal service deployment across multiple clouds,” *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1431–1441, 2013.

[26] S. Sotiriadis, N. Bessis, P. Kuonen, and N. Antonopoulos, “The Inter-Cloud Meta-Scheduling (ICMS) Framework,” in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, 2013, pp. 64–73.

[27] ETSI Industry Specification Group, “NFV Management and Orchestration,” 2014.

[28] X. Wang, X. Fu, X. Liu, and Z. Gu, “Power-aware cpu utilization control for distributed real-time systems,” pp. 233–242, April 2009.

[29] L. Tomás, C. Klein, J. Tordsson, and F. Hernández-Rodríguez, “The straw that broke the camel’s back: Safe cloud overbooking with application brownout,” pp. 151–160, Sept 2014.

[30] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, and T. LaPorta, "Fast and robust signaling overload control," in *IEEE Intl. Conf. Network Protocols,* 2001.

[31] M. Welsh and D. E. Culler, "Adaptive Overload Control for Busy Internet Servers," in *USENIX Symposium on Internet Technologies and Systems,* 2003.

[32] M. Fowler, *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[33] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Computing Surveys (CSUR),* vol. 51, no. 2, p. 40, 2018.

[34] L. Chi and X. Zhu, "Hashing techniques: A survey and taxonomy," *ACM Computing Surveys (CSUR),* vol. 50, no. 1, p. 11, 2017.

[35] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking (TON),* vol. 11, no. 1, pp. 17–32, 2003.

[36] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse, "Characterizing load imbalance in real-world networked caches," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks.* ACM, 2014, p. 8.

[37] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 13.

[38] B. Oonhawat and N. Nupairoj, "Hotspot management strategy for real-time log data in mongodb," in *Advanced Communication Technology*

(ICACT), 2017 19th International Conference on. IEEE, 2017, pp. 221–227.

[39] X. Liu, J. Heo, L. Sha, and X. Zhu, “Adaptive control of multi-tiered web applications using queueing predictor,” in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP.* IEEE, 2006, pp. 106–114.

[40] L. Cherkasova and P. Phaal, “Session-based admission control: A mechanism for peak load management of commercial web sites,” *IEEE Transactions on computers*, vol. 51, no. 6, pp. 669–685, 2002.

[41] S. Muppala and X. Zhou, “Coordinated session-based admission control with statistical learning for multi-tier internet applications,” *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 20–29, 2011.

[42] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, “Autoscale: Dynamic, robust capacity management for multi-tier data centers,” *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, p. 14, 2012.

[43] G. C. Salvatore Dipietro, “Pax: Partition-aware autoscaling for the cassandra nosql database,” in *Network Operations and Management Symposium, 2018. NOMS 2018. 30th IEEE/IFIP.* IEEE, 2018.

[44] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems.* John Wiley & Sons, 2004.

[45] Y. Hong, C. Huang, and J. Yan, “A comparative study of SIP overload control algorithms,” *Network and traffic engineering in emerging distributed computing applications*, 2012.

[46] Project Clearwater. [Online]. Available: http://www.projectclearwater.org/

[47] A. Manzalini, R. Minerva, E. Kaempfer, F. Callegari, A. Campi, W. Cerroni, N. Crespi, E. Dekel, Y. Tock, W. Tavernier *et al.*, "Manifesto of edge ICT fabric," in *Proc. ICIN*, 2013, pp. 9–15.

[48] R. Jain and K. K. Ramakrishnan, "Congestion Avoidance in Computer Networks with a Connectionless Network Layer, Part I: Concepts, Goals and Methodology," in *Proc. Computer Networking Symposium*, 1987, pp. 134–143.

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

[50] NFV ISG, "Network Function Virtualisation (NFV) - Resiliency Requirements," ETSI, Tech. Rep., 2014.

[51] ——, "Network Function Virtualisation (NFV) - Use Cases," ETSI, Tech. Rep., 2013.

[52] ——, "Network Functions Virtualisation (NFV) - Management and Orchestration," ETSI, Tech. Rep., 2014.

[53] ——, "Network Function Virtualisation Infrastructure Architecture - Overview," Tech. Rep., 2014. [Online]. Available: http://docbox.etsi.org/ISG/NFV/Open/Latest_Drafts/nfv-inf001v036-InfrastructureOverview.pdf

[54] S. Van Rossem, B. Sayadi, L. Roullet, A. Mimidis, M. Paolino, P. Veitch, B. Berde, I. Labrador, A. Ramos, W. Tavernier *et al.*, "A vision for the next generation platform-as-a-service," in *2018 IEEE 5G World Forum (5GWF)*. IEEE, 2018, pp. 14–19.

[55] H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. Van Rossem, W. Tavernier *et al.*, "Devops for network function virtualisation: an architectural approach," *Transactions*

on Emerging Telecommunications Technologies, vol. 27, no. 9, pp. 1206–1215, 2016.

[56] B. Berde, S. Van Rossem, A. Ramos, M. Orrù, and A. Shatnawi, “Dev-for-operations and multi-sided platform for next generation platform as a service,” in *2018 European Conference on Networks and Communications (EuCNC)*. IEEE, 2018, pp. 1–5.

[57] CloudWatch, Amazon. Creating amazon cloudwatch alarms. [Online]. Available: http://docs.aws.amazon.com/AmazonCloudWatchEvents/latest/APIReference/Welcome.html

[58] StackDriver, Google. Stackdriver monitoring. [Online]. Available: https://cloud.google.com/monitoring/

[59] Datadog. Introducing anomaly detection in datadog. [Online]. Available: https://www.datadoghq.com/blog/introducing-anomaly-detection-datadog/

[60] J. Zhang, M. Zulkernine, and A. Haque, “Random-forests-based network intrusion detection systems,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 5, pp. 649–659, 2008.

[61] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, “A geometric framework for unsupervised anomaly detection,” in *Applications of data mining in computer security*. Springer, 2002, pp. 77–101.

[62] J. Zhang and M. Zulkernine, “Anomaly based network intrusion detection with unsupervised outlier detection,” in *Communications, 2006. ICC’06. IEEE International Conference on*, vol. 5. IEEE, 2006, pp. 2388–2393.

[63] F. Schmidt, A. Gulenko, M. Wallschläger, A. Acker, V. Hennig, F. Liu, and O. Kao, "Iftm-unsupervised anomaly detection for virtualized network function services," in *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 2018, pp. 187–194.

[64] G. Jakobson and M. Weissman, "Alarm correlation," *IEEE Network*, vol. 7, no. 6, pp. 52–59, Nov 1993.

[65] A. T. Bouloutas, S. Calo, and A. Finkel, "Alarm correlation and fault identification in communication networks," *Communications, IEEE Transactions on*, vol. 42, no. 234, pp. 523–533, Feb 1994.

[66] F. Cuppens and A. Miege, "Alert correlation in a cooperative intrusion detection framework," in *Security and Privacy, 2002 IEEE Symposium on*, 2002, pp. 202–215.

[67] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "A coding approach to event correlation," in *Integrated Network Management IV*. Springer, 1995, pp. 266–277.

[68] S. H. Nikounia *et al.*, "Hypervisor and neighbors' noise: Performance degradation in virtualized environments," *IEEE Transactions on Services Computing*, 2015.

[69] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[70] F. Caglar, S. Shekhar, and A. Gokhale, "A performance interference-aware virtual machine placement strategy for supporting soft realtime applications in the cloud," *Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA, Tech. Rep. ISIS-13-105*, 2013.

[71] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, 2017, pp. 71–84.

[72] A. O. Ayodele, J. Rao, and T. E. Boult, "Performance measurement and interference profiling in multi-tenant clouds," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on.* IEEE, 2015, pp. 941–949.

[73] P. Hofer, F. Hörschläger, and H. Mössenböck, "Sampling-based steal time accounting under hardware virtualization," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering.* ACM, 2015, pp. 87–90.

[74] M. Yamamoto and K. Kohta Nakashima, "Execution time compensation for cloud applications by subtracting steal time based on host-level sampling," in *Companion Publication for ACM/SPEC on International Conference on Performance Engineering.* ACM, 2016, pp. 69–73.

[75] S. Team. (2013) Understanding CPU Steal Time - when should you be worried? [Online]. Available: http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when\-should-you-be-worried

[76] VMware, Inc. (2008) Timekeeping in VMware virtual machines. [Online]. Available: http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf

[77] G. Casale, C. Ragusa, and P. Parpas, "A feasibility study of host-level contention detection by guest virtual machines," in *Cloud Computing Tech-*

nology and Science (CloudCom), 2013 IEEE 5th International Conference on, vol. 2. IEEE, 2013, pp. 152–157.

[78] P. Leitner and J. Scheuner, “Bursting with possibilities–an empirical study of credit-based bursting cloud instance types,” in *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE, 2015, pp. 227–236.

[79] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[80] D. Cotroneo, R. Natella, and S. Rosiello, “Nfv-throttle: An overload control framework for network function virtualization,” *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 949–963, 2017.

[81] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. H. Huang, “Load balancing of heterogeneous workloads in memcached clusters.” in *Feedback Computing*, 2014.

[82] W. Zhang, T. Wood, and J. Hwang, “Netkv: Scalable, self-managing, load balancing as a network function,” in *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 5–14.

[83] Y. Cheng, A. Gupta, and A. R. Butt, “An in-memory object caching framework with adaptive load balancing,” in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 4.

[84] R. Taylor, “Interpretation of the correlation coefficient: a basic review,” *Journal of diagnostic medical sonography*, vol. 6, no. 1, pp. 35–39, 1990.

[85] D. Rumsey, “How to interpret a correlation coefficient r,” *Statistics For Dummies*, 2016.

[86] G. F. Reed, F. Lynn, and B. D. Meade, "Use of coefficient of variation in assessing variability of quantitative assays," *Clinical and diagnostic laboratory immunology*, vol. 9, no. 6, pp. 1235–1239, 2002.

[87] ETSI 3GPP, "Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; IP Multimedia Subsystem (IMS); Stage 2," ETSI, Tech. Rep., 2013.

[88] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, 2004.

[89] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[90] K. Jackson, *OpenStack Cloud Computing Cookbook.* Packt Publishing, 2012.

[91] Timekeeping Virtualization for X86-Based Architectures. [Online]. Available: https://www.kernel.org/doc/Documentation/virtual/kvm/timekeeping.txt

[92] Timekeeping in VMware Virtual Machines. [Online]. Available: http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf

[93] F. E. Goncalves and B.-A. Iancu, *Building Telephony Systems with OpenSIPS.* Packt Publishing Ltd, 2016.

[94] R. Gayraud, O. Jaques *et al.*, "SIPp: SIP load generator," 2010. [Online]. Available: http://sipp.sourceforge.net/

[95] R. Redelmeier, "CPUburn — CPU testing utility," 2010. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man1/cpuburn.1.html

[96] R. L. Freeman, *Telecommunication system engineering.* John Wiley & Sons, 2015, vol. 82.

[97] E. Stahl, A. Corona, F. De Gilio *et al.*, *Performance and Capacity Themes for Cloud Computing.* IBM Redbooks, 2013.

[98] Red Hat Inc. (2014) Virtualization deployment and administration guide. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/index

[99] C. Ehrhardt. (2010) CPU time accounting. [Online]. Available: http://public.dhe.ibm.com/software/dw/linux390/perf/CPU_time_accounting.pdf

[100] R. van der Heij. (2014) Understanding Linux on z/VM steal time. [Online]. Available: http://www.velocitysoftware.com/VMSTEAL.PDF

[101] VMware Inc. (2016) Guest and HA Application Monitoring Developer's Guide. [Online]. Available: https://pubs.vmware.com/vsphere-6-0/topic/com.vmware.ICbase/PDF/vs600_guest_HAappmon_sdk.pdf

[102] A. Lê-Quôc, M. Fiedler, and C. Cabanilla. (2013) The top 5 AWS EC2 performance problems. [Online]. Available: https://www.datadoghq.com/blog/top-5-ways-to-improve-your-aws-ec2-performance/

[103] N. Shestakov. (2016) Hyper-V performance analysis with Veeam ONE. [Online]. Available: https://hyperv.veeam.com/blog/how-to-analyse-visualise-hyper-v-performance/

[104] V. Hilt and I. Widjaja, "Controlling Overload in Networks of SIP Servers," in *Network Protocols, 2008. ICNP 2008. IEEE International Conference on.* IEEE, 2008, pp. 83–93.

[105] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *IEEE Intl. Conf. Dependable Systems and Networks*, 2000.

[106] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery From Software Failures Caused by Mandelbugs," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 70–87, 2016.

[107] C. Shen, H. Schulzrinne, and E. M. Nahum, "Session Initiation Protocol (SIP) server overload control: Design and evaluation," *IPTComm*, vol. 8, pp. 149–173, 2008.

[108] J. Mo, R. J. La, V. Anantharam, and J. Walrand, "Analysis and comparison of TCP Reno and Vegas," in *Annual Joint Conference IEEE Computer and Communications Societies (INFOCOMM)*, 1999.

[109] K. Ramakrishnan and R. Jain, "A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer," in *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, 1988, pp. 303–313.

[110] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman, "One more bit is enough," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 37–48, 2005.

[111] D. Cotroneo, R. Natella, and S. Rosiello, "NFV-Throttle: An overload control framework for Network Function Virtualization," *IEEE Transactions on Network and Service Management*, 2017.

[112] L. Eggert and J. D. Touch, "Idletime scheduling with preemption intervals," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005, pp. 249–262.

[113] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, “The Linux scheduler: A decade of wasted cores,” in *11th ACM European Conference on Computer Systems*, 2016.

[114] Xen Project. (2017) Credit scheduler. [Online]. Available: https://wiki.xen.org/wiki/Credit_Scheduler

[115] J. Corbet. (2002) A safe SCHED_IDLE implementation. [Online]. Available: https://lwn.net/Articles/4073/

[116] Windows Dev Center. (2017) Task Idle Conditions. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa383561(v=vs.85).aspx

[117] L. Torvalds et al. (2007) Linux kernel parameters. [Online]. Available: https://www.kernel.org/doc/Documentation/admin-guide/kernel-parameters.txt

[118] Metaswitch Networks Ltd., “Clearwater SIP Stress scenario,” 2016. [Online]. Available: https://github.com/Metaswitch/sprout/blob/dev/clearwater-sip-stress.root/usr/share/clearwater/sip-stress/sip-stress.xml

[119] J. Heo and L. Singaravelu, “Deploying extremely latencysensitive applications in vsphere 5.5: Performance study,” *VMware, Inc., Tech. Rep*, 2013.

[120] D. M. Davis, “Demystifying cpu ready (%rdy) as a performance metric: Don’t trust available cpu,” *Quest Software, White paper*, 2013.

[121] D. Cotroneo, L. De Simone, and R. Natella, “NFV-Bench: A dependability benchmark for Network Function Virtualization systems,” *IEEE Transactions on Network and Service Management*, 2017.

[122] A. Beloglazov and R. Buyya, “Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1366–1379, 2013.

[123] N. F. V. ETSI, “Use cases,” *GS NFV*, vol. 1, 2013.

[124] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri, “Robust estimation of resource consumption for SQL queries using statistical techniques,” *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1555–1566, 2012.

[125] ETSI Industry Specification Group, “Network Functions Virtualisation: Infrastructure Overview,” 2015.

[126] T. A. Foundation. (2018) Apache Cassandra. [Online]. Available: http://cassandra.apache.org/

[127] Clearwater performance and our load monitor. [Online]. Available: http://www.projectclearwater.org/clearwater-performance-and-our-load-monitor/

[128] Tuning overload control for telco-grade performance. [Online]. Available: http://www.projectclearwater.org/overload-control-2/

[129] M. Rajashekhar, “Caching at twitter and moving towards a persistent, in-memory key-value store,” 2012. [Online]. Available: http://cloudseminar.berkeley.edu/data/twittercache.pdf

[130] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, “Scaling memcache at facebook,” in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation.* USENIX, 2013, pp. 385–398.

[131] J. T. T. Griffith, “Optimizing caching: Twemproxy and memcached at flickr,” 2015. [Online]. Available: http://code.flickr.net/2015/07/10/optimizing-caching-twemproxy-and-memcached-at-flickr/