SAPIENZA
UNIVERSITÀ DI ROMA

FACULTY OF INFORMATION ENGINEERING, COMPUTER SCIENCE AND STATISTICS
DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

PHD IN ENGINEERING OF COMPUTER SCIENCE

XXX CYCLE

Autoscaling Techniques and Blockchain-based Architectures for Performant and Dependable Complex Distributed Systems

AUTHOR: Federico Lombardi

ADVISOR: Silvia Bonomi
CO-ADVISOR: Giorgio Grisetti
ASSISTANT ADVISOR: Leonardo Aniello

ACADEMIC YEAR
2016-2017

Autoscaling Techniques and Blockchain-based Architectures for Performant and Dependable Complex Distributed Systems
Ph.D. Thesis

Department of Computer, Control, and Management Engineering "Antonio Ruberti"
Sapienza University of Rome
Via Ariosto 25, I-00185 Rome, Italy
lombardi@dis.uniroma1.it
http://www.dis.uniroma1.it/~dottoratoii/students/302.php

To my Family, who always believed in me.
To my Mom, who will always be in me.

Acknowledgements

I believe that the greatest thing in achieving a goal is to share it with people you love. I really wish to thank my family with all my heart, especially my daddy, my grandparents and Benedetta for always supporting me and helping me to choose the right way. A special thanks goes to Silvia, Leo A., Leo Q. and Roberto, who introduced and drove me in a fabulous experience and made me grow up and discover the world of the research. Thank you for being always available and for your friendship.

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

Abstract

Dependability is an important measure which combines several properties to draw up the quality of a system. It is often combined with *performability* which quantifies how well the system performs in the presence of faults over a specified period of time. The recent wide spread of Cloud Computing has made such properties paramount, however as modern systems are increasing their complexity, ensuring such properties has become harder.

In this thesis we study solutions to improve performability and dependability of modern complex distributed systems. The thesis is divided in two parts.

In the first part we evaluate automatic scaling techniques to elastically reconfigure the number of resources of a distributed system according to workload changes. We started with the claim that proactivity may lead to a more efficient usage of resources; so we design MYSE, an architecture to automatically scale a distributed system proactively according to workload changes. The promising results obtained from simulations carried us to propose a refinement of MYSE for real-world scenarios, so we designed PASCAL. As a first case study we proposed a solution for a distributed datastore that we tested on a real prototype within the Cassandra datastore. Similarly, we proposed ELYSIUM, an instantiation of PASCAL for stream processing systems which symbiotically combines scaling of operators and resources. We integrated ELYSIUM within the scheduler of a well-known stream processing engine, namely Storm. The obtained results for both case studies backup our initial claim, showing as proactivity may improve performances while saving more resources, since it is possible to reduce under/over-provisioning periods.

In the second part we assess architectures based on blockchain for tamper-resistant multi-party interactions. We started introducing the SUNFISH project and the design of the Federated Service Ledger infrastructure. Then, we propose SLAVE, a solution which exploits blockchain for log-base disputes resolution among parties. Following, we stated some open research questions to adopt a blockchain as a distributed database, thus we proposed 2LBC, a two-layer architecture that relies on a permissioned blockchain anchored to a permissionless one to ensure high data integrity without sacrificing performances. An evaluation of a real prototype shows the effectiveness of our approach, but highlights how the distributed consensus plays a prominence role for both performance and security of a blockchain-based system. Thus, we evaluated a new family of consensus algorithms designed for blockchain, i.e. Proof-of-Authority, to properly assess which family can offer more guarantees. Through a CAP theorem-based analysis we compared their guarantees with PBFT, figuring out that so far choosing a consensus is strictly dependent from the target scenario.

Finally, we concluded the work by discussing a possible integration of the two parts of the thesis to design a decentralised elastic dependable framework.

Contents

Introduction

Nowadays, the increasing need of high computation performance in Data Center is fostering the wide employment of services distributed on multiple hosts. Those systems allow to sustain higher amounts of workload, improve performances and avoid single point of failure. The recent spread of Cloud Computing is proof of success of such systems which make the *elasticity* a key feature. They can indeed have virtually an infinite amount of resources to rent to user services like physical machines, virtual machines (VMs), software and turn-on/off new resources only when necessary. The user is charged with a *pay-as-you-go* price model, thus both providers and users can minimise their costs by effectively tuning the resource to use. Moreover, the Cloud paradigm permits to simplify the hardware of final user devices as a Cloud provider is delegated for computation and storage of user data. A good management of Cloud might bring a number of advantages for both providers and final users beyond an economical perspective.

Although moving from a private system to a Cloud-based one may lead to huge benefits, the complexity of the system increases and a number of open research problems need to be addressed. For instance, an user who pays a provider for a service claims a Quality-of-Service (QoS) like a specified maximum response time or a maximum percentage of unavailability. Such a QoS is defined often as a formal contract between provider and user, namely Service Level Agreement (SLA), which the provider has to guarantee. A violation of such SLAs may lead to huge economic harm for providers and loss of reputation. Thus, to provide high reliable and performant services, providers need to design systems able to cope with tough conditions like increasing workload, errors, faults and so on, which may undermine performance and reliability of the system.

Furthermore, from a security perspective, the user has to trust the Cloud provider as it is in control of computation and storage of her data. Such a trust may hinder a wide employment of Cloud when data are sensitive. Therefore, in order to safely delegate the management of sensitive data to a third party, the user urgently needs new solutions for reliable auditability.

Dependability is the measure of how a system is able to tackle possible tough conditions. Recently it is often combined with *performability*, as they are strictly related and one can affect the other. An ideal *dependable* and *performant* system should be resilient to a number of benign and malicious threats ranging from availability to security.

Thesis Intention and Structure

The goal of this thesis is to study and evaluate techniques to achieve and improve both performability and dependability of complex modern distributed systems. In Chapter 1 we outline the context, by introducing dependability in distributed systems and Cloud Computing scenarios, while in Chapter 2 we point out how to assess such systems.

Then, we structure the contributions in two parts, each one facing different dependable attributes. In Part I we target the *availability* while in Part II we focus on the *security* (specifically, on the *data integrity*). In particular, in Part I we assess automatic (or elastic) scaling techniques to maximise performances and ensure system availability while minimising the costs; whereas in Part II we evaluated architectural patterns based on blockchain to ensure data integrity without sacrificing performances.

Besides dependable attributes, we consider different Cloud deployment models for the two parts. Indeed, in Part I we propose optimisations for private and public Cloud, whereas in Part II we discuss about performance and security issues of Cloud federations.

The two parts, moreover, differ for the level of trust among nodes composing the distributed system. Indeed, in a private/public Cloud scenario we can consider high trust between nodes being part of the same system. In the second part, coping with multi-party federations, we consider low trust among federated members, hence we deal with possible subverted nodes which may behave maliciously. Finally, a further Part III sums up the thesis and discusses ongoing work and future research directions.

Structure and Content of Part I

In Chapter 3 we start with the claim that proactivity may lead benefits due to a more efficient resource usage and so we design a proactive auto-scaling architecture, namely MYSE. Then, in Chapter 4 we refine such an architecture in order to make it employable in a number of real world scenarios, thus we propose PASCAL. In Chapter 5 we propose an instantiation of PASCAL for a distributed datastore. Finally. in Chapter 6 we propose ELYSIUM, an elastic scaling solution for stream processing systems.

Structure and Content of Part II

In Chapter 7 we introduce the context of multi-party transactions. We also detail SUNFISH, an EU project aimed to design secure Cloud federations that we consider as a reference scenario for the rest of the second part of the thesis and a solution for one of its use case. In Chapter 8 we propose SLAVE, a solution to solve disputes in multi-party transactions. In Chapter 9 we investigate how to design a blockchain-based database. Finally, as *distributed consensus* plays a key role in a blockchain, in Chapter 10 we study new democratic schemas to achieve agreement.

Contributions

The contributions for the first part can be summarised follow:

- we propose MYSE [9], an architecture for automatic scaling a replicated service which combine Artificial Neural Network for workload forecasting with a queuing model for performance estimation and a heuristic to minimise oscillations;
- we propose PASCAL, a refinement of MYSE designed to addressed several real world systems and we provide a case study on a real distributed datastore, i.e. Cassandra;
- we propose ELYSIUM [153], an elastic scaling solution for stream processing systems which symbiotically scale operators and resources and we provide an evaluation of a real prototype integrated within a real Stream Processing engine, namely Storm.

The contributions for the second part can be summarised follow:

- we propose Federated Service Ledger as part of the underlying SUNFISH infrastructure and designed a blockchain-based solution for a payslip use case [179];
- we propose SLAVE [8], an architecture and a solution to solve disputes in multi-party transactions through a blockchain;
- we introduce a few open research questions to adopt a blockchain as a distributed database and we propose 2LBC [89, 7], an architecture based on two layers of blockchain to ensure data integrity and high performance;
- we assess a new family of consensus algorithms named Proof-of-Authority and examine through a reverse engineering of the source code of two main Ethereum implementations the message pattern exchange so as to evaluate performances, and through the CAP theorem analyse their security [215].

Chapter 1

Background and Context

1.1 Distributed Systems and Dependability

In the field of distributed systems, the *dependability* is a measure which encloses a set of characteristics that a system has to provide to be trustworthy within a time period [20, 18]. As dependability is strictly related to some security aspects, it is common referred as *Dependability and Security*. It can be mainly broken down in three elements:

- *Attributes*: a way to asses the system dependability;
- *Threats*: main issue which might affect the system dependability;
- *Means*: solutions to improve the system dependability.

1.1.1 Attributes

Attributes represent the main properties of a system that can be assessed its dependability level by using both qualitative and quantitative measures. Main attributes for a dependable system can be summarised as follow:

- *Availability*: ability of the system to run without interruption;
- *Reliability*: ability of the system to run continuously providing a correct service without failures;
- *Maintainability*: disposition of the system to be repaired;
- *Safety*: a system which can ensure absence of catastrophic consequences to users and environment;
- *Security*: ability of the system to provide *confidentiality* (i.e. avoiding leaking of sensitive information) and *integrity* (i.e. access and alteration of data only by authorised users and according to allowed mode).

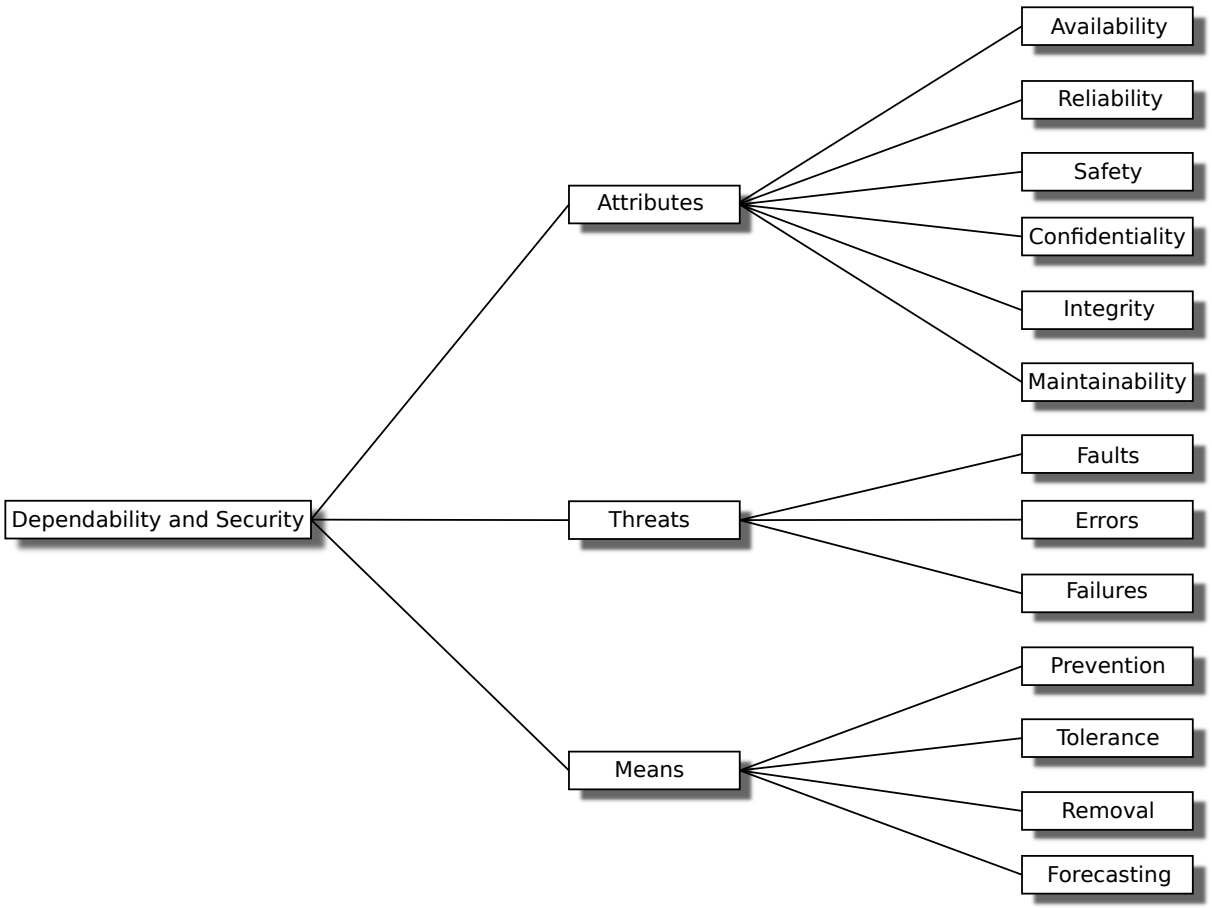


Figure 1.1: *Taxonomy of Dependability & Security with Attributes, Threats and Means*

Although several definition of the above properties have been proposed, the most common formalisation is provided by Avizienis et al. [20, 18], and by extending them with a *Performability* property, defined as *quantifying how well the object system performs in the presence of faults over a specified period of time* [165].

Specifically, the *performability* is considered as the ability of the system to provide a service in a desired time. Such a property is mostly related to the dynamism of the system, as it is a tricky task to satisfy runtime with dynamic workload. This aspect will be explained in the next section, describing issues and challenges of Scalability and Elasticity of a system under dynamic workload.

1.1.2 Threats

Threats of a dependable system, are issues which might affect the system, causing a drop in dependability. These issues have been mainly categorised in three terms which can lead each other [20, 18, 19]. Specifically, they can be categorised as follow:

- *Fault*: A fault (which is usually referred to as a bug for historic reasons) is a defect in a system. The presence of a fault in a system may or may not lead to a failure. For instance, although a system may contain a fault, its input and state conditions may never cause this fault to be executed so that an error occurs; and thus that particular fault never exhibits as a failure.

- *Error*: An error is a discrepancy between the intended behaviour of a system and its actual behaviour inside the system boundary. Errors occur at runtime when some part of the system enters an unexpected state due to the activation of a fault. Since errors are generated from invalid states they are hard to observe without special mechanisms, such as debuggers or debug output to logs.

- *Failure*: A failure is an instance in time when a system displays behaviour that is contrary to its specification. An error may not necessarily cause a failure, for instance an exception may be thrown by a system but this may be caught and handled using fault tolerance techniques so the overall operation of the system will conform to the specification.

Note that Faults, Errors and Failures are related according to a flow known as a *Fault-Error-Failure chain* [20].

As a rule of thumb, a Fault can lead to an Error which brings to an invalid state, hence it can lead to other errors or to a failure. A Failure is an observable deviation from the normal behaviour at the system boundary, so they can be recorded due to propagated errors.

As output data from a service can be fed into another service, a failure in a service may propagate into another one as a fault, and so on by leading to a chain in the form: $Fault \rightarrow Error \rightarrow Failure \rightarrow Error \cdots$.

1.1.3 Means

In response to the Fault-Error-Chain, it is possible to design *means* in order to break the chain with the aim to improve the dependability of a system. Four approaches have been proposed:

- *Fault Prevention*: deals with preventing faults being incorporated into a system. This can be accomplished by use of development methodologies and good implementation techniques.

- *Fault Removal* which, in turn, can be divided in two sub-categories:

  - *Removal During Development*: requires verification so that faults can be detected and removed before a system is put into production;
  - *Removal During Use*: once a system has been put into production, it has to record failures and remove them through a maintenance cycle.

- *Fault Forecasting*: in order to remove and/or circumvent their affects, faults are predicted.

- *Fault Tolerance*: deals with putting mechanisms in place that will allow a system to still deliver the required service in the presence of faults, although that service may be at a degraded level.

Essentially, dependability means are techniques employed to cope with failures so as to increase the experience of the user into the system. All these approach as well as the classical dependability concept have been designed for a static universe and small scale. Figure 1.1 shows the taxonomy explained in this section.

1.2 Dynamic Dependable Systems

1.2.1 Workload Pattern

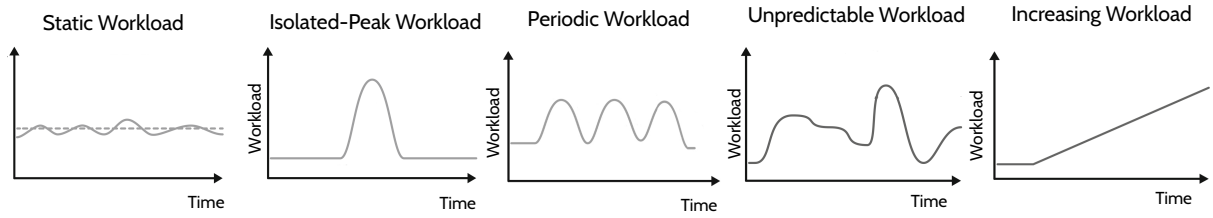


Figure 1.2: *Workload Patterns*

Understanding the characteristics and patterns of workloads is critical in order to conduct an effective Capacity Planning process and thus optimising and improving the performance of current computing systems, improve the resource management and at the same time provide Quality of Service guarantees. Once workload patterns are investigated, it is equally important to possess the ability to predict how such workloads will vary in the future; to this end Workload Forecasting techniques may be adopted. This is especially important in the Cloud Computing environment, where effectively exploiting the elasticity of a computing system, it is possible to adapt the provisioned resources in a timely manner such that the application will be always able to to handle the imposed workload while at the same time maximise the elasticity effectiveness of the whole system, reducing resource inactivity.

Workload Patterns describe different types of user interaction with the computing system that result in different changes in the utilisation of system resources. The workload can be measured in terms of user requests submitted to the system per time unit, server utilisation, network traffic etc. Learning and classifying workload patterns is important in order to design effective scaling policies and select adequate workload forecasting techniques. Figure 1.2 shows possible and most common workload patterns, i.e.:

- *Static Workloads*: are characterized by an average flat utilisation of the computing system resources over time. Therefore, there is normally no necessity to vary the amount of provisioned resources in order to adapt to workload changes. When provisioning for this type

of workload a constant number of resources is required, plus an optional little amount of over-provision in order to deal with minimal variations of workload. For what concerns the Cloud environment, static workload are clearly less likely to take advantage from the Cloud Elasticity and the pay-per-use pricing policy.

- *Isolated-Peak Workloads*: are characterized by an average flat utilisation of computing resources disturbed by an isolated strong peak. The occurrence of this peak may be known in advance as it may be correlated to specified events. Isolated-Peak workload may be handled with the same automated mechanisms as periodic workload. However, in contrast to these automated mechanisms, the resource provisioning and releasing tasks necessary for this workload may also be handled manually using the self-service capability of the used cloud offerings. As resource provisioning and releasing is only performed once, the benefits of an automated alignment of IT resource numbers to the experienced workload are reduced possibly making the additional effort to automate them unreasonable. Therefore, it may be a better choice to handle this situation by human intervention.

- *Periodic Workloads*: are characterized by peak and non-peak resource utilisation periods occurring at periodic time intervals. They are experienced by computing systems whose users interact with at well-defined time intervals that recur over time. In a static provisioning context, provisioning for a periodic workload implies over-provision the system with enough resources to handle higher resource utilisation peaks in the periodic pattern, resulting in a resource waste in non-peak periods of the pattern. A periodic pattern can greatly take advantage of the Cloud environment and its pay-per-use pricing policy, since it is possible to elastically provision resources during the peaks and releasing them as soon as the peaks are ended.

- *Unpredictable Workloads*: are characterized by a random, non-predictable computing resource demand. They necessarily require unplanned and unpredictable provisioning and release of resources, and therefore it is extremely hard to handle provide an effective static scaling solution. Cloud computing elasticity can help to adequately handle this type of workload, but since it is not possible to rely on any workload prediction technique, efficiency of elastic scaling actions is necessarily reduced.

- *Increasing/Decreasing Workloads*: are characterized by a continuous constant growth or decline in the utilisation of computing resources. A long-term increasing workload often corresponds to the growth of a successful business application after its launch. Decreasing workload is often experienced by legacy application that still handle some workload that is slowly fading away. In both cases, computing resources need to be provisioned or released to match the workload intensity. The growing or shrinking rate can be known in advance or not, and can be linear, non-linear, exponential etc., but in any case the workload change is consistent in the same direction.

1.2.2 Scalability

The scalability of an application or computing system is a measure of the number of users it can effectively support at the same time while maintaining an adequate performance level [35]. The point at which an the application or computing system cannot effectively handle additional users is the limit of its scalability. Such limit is reached when a critical hardware resource runs out, and in order to overcome such limit it is typically required to provide additional hardware resources like CPU, memory, disk, and network bandwidth, as long as the application or system can effectively utilise those resources [233].

There are two different approaches, often referred to as scaling dimensions, that can be taken in order to provide additional resources or to release them from the application or computing system:

- *Vertical Scaling* (Scale Up/Down): it means to add/remove resources to a given system node, like CPU cores or memory in a way that the system node can handle a larger/smaller workload. However, the achievable improvements are limited by the extent to which the software is able to take advantage of the hardware. Because hardware changes are involved, usually this approach involves downtime;
- *Horizontal Scaling* (Scale Out/In): it means to add/remove system nodes (e.g. virtual machine instances or physical machines) to a distributed system in a way that the entire system can handle bigger/smaller workloads. Depending on the type of the application, especially when dealing with shared data, scaling out the system often implies an increase in communication overheads and prevent the emergence of substantial performance gains, especially when adding nodes at bigger cluster sizes. In some scenarios, scaling horizontally may even result in performance degradation. Horizontal scaling is limited by the efficiency of added nodes. The best outcome is when each additional node adds the same incremental amount of usable capacity.

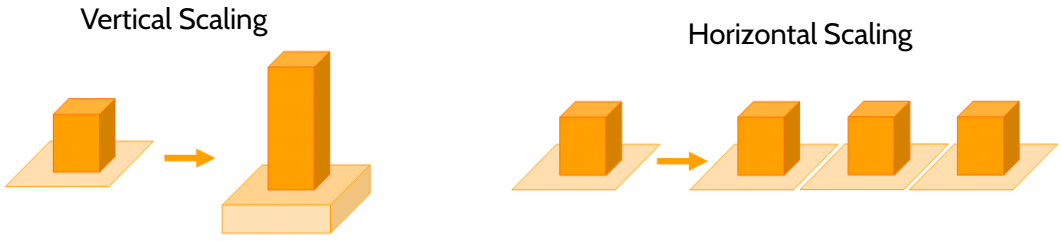


Figure 1.3: *Vertical and Horizontal Scaling*

Figure 1.3 shows the two scaling approaches. There are trade-offs between the two approaches. Scaling Horizontally increasing the number of nodes of the distributed system may increase the system management complexity, as well as a more complex programming model; also, only a subclass of applications can be deployed in a distributed environment. In the past, the price difference

between the two approaches has favored the vertical scaling, but recent advances in virtualization technology and the new cloud computing paradigm have blurred that price advantage, since deploying a new virtual system over a virtual platform is often less expensive than actually buying, installing and configuring new hardware. It is important to notice that this definition of scalability does not include any temporal aspect: scaling means that resources are provisioned or released according to a predefined approach, but it does not specify how fast, how often and how significantly the needed resources are provisioned.

1.2.3 Elasticity

Elasticity is the degree of a system to adapt to load changes by automatically provisioning and releasing resources in order to offer at each temporal instant an amount of available resources which matches the current demand as closely as possible [115]. An elastic system is so able to change its own configuration through scaling actions which trigger provisioning/de-provisioning of resources from the system itself. Such scaling actions are triggered runtime and according to different scaling policies, so as to dynamically adapt the configuration of the system to resource demand. The actual effects of scaling actions mainly depend on (i) the current state, (ii) previous state of the system and (iii) application running on the physical infrastructure. Consequently, elasticity can be considered as a multi-dimentional metric which depends from temporal and quantitative dimensions [139].

The temporal dimension of elasticity can be modeled with the notions of *Demand Point*, *Triggering Point* and of *Reconfiguration Point*. The Demand Point (DP) is the point in time at which the resource demand of the system changes with respect to the current configuration. The Triggering Point (TP) is the point in time at which the system triggers the execution of the scaling action in order to change its configuration such that the new available resources are equal to the resource demand at the Demand Point. The Reconfiguration Point (RP) is a point in time at which the system adaptation that follows the configuration change triggered by the scaling action (resource provisioning or release) is completely processed by the system. Note that a reconfiguration point also takes into account that the effects of the system reconfiguration may become visible some time after the scaling action completes its execution, since the system may need time to adapt to the changed resource availability. Furthermore, it is important to know that while reconfiguration points and the time point of visibility of effects may be measurable, the triggering points may not be directly observable.

The quantitative dimension of elasticity can be modeled with the notions of Over-provisioning Penalty, Under-provisioning Penalty, Total Penalty. The Over-provision Penalty (OPP) is time difference between RP and DP times the difference between the amount of resources provisioned at RP and the amount of resources required at DP. It measures for how much time the system is provisioned with more resources that what is actually required, and the amount of such resources.

$$OPP = (RP - DP) \cdot (Resources(RP) - Resources(DP))$$

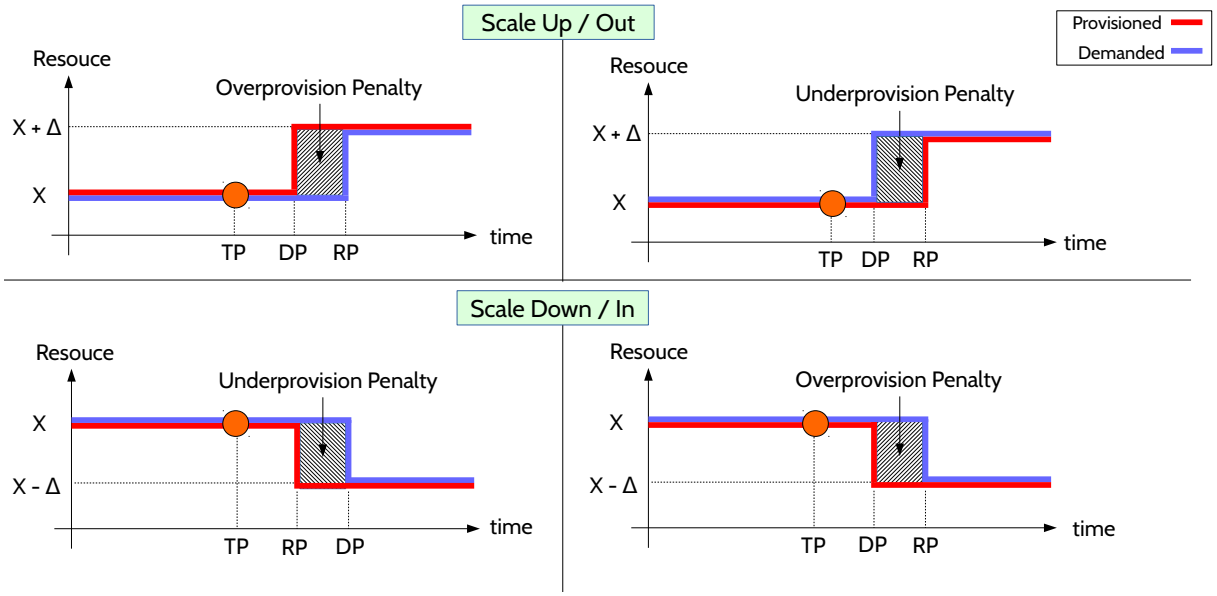


Figure 1.4: *Elasticity as a metric dependent from over/under-provisioning periods*

The Under-provisioning Penalty (UPP) is time difference between DP and RP times the difference between the amount of resources required at DP and the amount of resources provisioned at RP. It measures for how much time the system is provisioned with less resources that what is actually required, and the amount of such resources.

$$UPP = (DP - RP)\dot{(}Resources(DP) - Resources(RP))$$

The Total Penalty (TPE) is the overall summation of Under-provisioning Penalties and Over-provisioning Penalties that occur during a given observation period.

$$TPE = \sum UPP_i + \sum OPP_i$$

Such penalty can be graphically imagined as the sum of the total grey area in Figure 1.4. Given the above definitions, elasticity effectiveness can be quantified claiming as the more Total Penalty tends to zero, the more the system effectively ensures its elasticity. Summarising, the elasticity reflects the sensitivity of a system's scaling process in relation to load intensity variations over time. Thus, scalability is a prerequisite for elasticity.

1.3 Cloud Computing

In the last years, Cloud Computing has emerged as a new information technology paradigm for managing and delivering services over the Internet. Cloud Computing users are provided with the possibility to deploy their applications in a large-scale environment with high scalability, availability, and fault tolerance characteristics, and at the same time reducing the administration costs.

The term Cloud Computing describes a category of different on-demand computing services offered by commercial providers. It denotes a model on which the computing infrastructure is viewed as a "Cloud", used by customers to access on demand applications from anywhere in the world. The main principle this model is based on is offering software, computing and storage capabilities "as a service" over the Internet. Although different definitions of Cloud Computing are present in the current literature (e.g. [40] [39] [13]), the mainly and broadly accepted is the one by NIST (National Institute of Standards and Technology) [163]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction

Main characteristics of the Cloud Computing environment are:

- *On-demand self-service*: the capability of a customer to provision computing and storage resources as and when they are needed automatically, without requiring human interaction with each service provider;
- *Broad network access*: network access through standard mechanisms to allow heterogeneous clients to make use of the resources;
- *Resource pooling*: the cloud provider's computing resources are pooled to serve multiple consumers in a multi-tenant model where free resources are dynamically assigned to subscribers requesting such resources according to consumer demand;
- *Rapid elasticity*: cloud resources can be dynamically provisioned and released, in some cases automatically, to rapidly scale outward and inward depending of the user demand. The cloud provider may provide an illusion of unlimited resources, so that the cloud customer may request for resources at any point in time.
- *Measured service*. Cloud resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilised service.

The advances in the virtualisation technology, united with the pay-as-you-go pricing model adopted by several cloud providers, have considerably changed the dynamics of infrastructure investment and deployment, reducing the initial capital expenses and making operational expenses predicable.

1.3.1 Cloud Service Models

Cloud Services may be classified into different Service Models depending on the type of services offered to the cloud customer:

- *Infrastructure-as-a-Service* (IaaS): such a service model provides the customer with the capability to provision processing, storage, networks, and other computing resources where the

consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage nor control the underlying physical cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g. firewalls). Distinct advantages of IaaS cloud model are the elimination of initial capital expense and a significant reduction in operating expenses, besides the advantages derived by the pay-as-you-go pricing model. Examples of IaaS cloud providers are Amazon AWS Elastic Compute Cloud (EC2) [75] and Google Compute Engine [79].

- *Platform-as-a-Service* (PaaS): such a service model provides the customer with the capability to deploy onto the cloud infrastructure custom or acquired applications created using development tools, programming languages, libraries and services supported by the cloud provider. The consumer does not manage nor control the underlying cloud infrastructure, but has full control over the deployed applications and possibly configuration settings for the hosting environment. Advantage of the PaaS cloud model is the simplification of application development and deployment by hiding the complexities of hardware and software resource management, and dynamic scaling of infrastructure based on the actual needs of the application. Example of PaaS cloud providers are Google App Engine (GAE) [78], Microsoft Azure [22] and IBM Bluemix [33].

- *Software-as-a-Service* (SaaS): such a service model provides the customer with the capability to directly use the provider's applications running on a cloud infrastructure. The applications are accessible through an heterogeneous set of client devices or program interfaces. The consumer does not manage nor control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings. Main advantages of SaaS cloud model are partial or full removal of need for client side software installation, the ubiquitous access and mandates less system requirements than in the traditional software service model. Examples of SaaS cloud providers are Google Apps [12] and Microsoft Office 365 [1].

1.3.2 Cloud Deployment Models

The implementation of a Cloud Computing infrastructure can be conducted according to one of the following deploying models.

- *Public Cloud*: the cloud infrastructure, typically huge interconnected data centers managed by the cloud provider, is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider. Public clouds are most suited for start-ups and small businesses because of minimal setup costs. The cloud resources are shared between multiple users. The infrastructure, services and usage policies

are managed by the service provider. The downside of public cloud deployment model is that the subscribers are generally not in control of the underlying hardware and software stacks involved in infrastructure management, as well as the location of data centers. Furthermore, there are security implications of multi-tenancy and country specific laws may impose restrictions on sensitive information crossing geographical borders. Examples of public clouds providers are Amazon Web Services [21], Rackspace [191], Google Cloud [54] and IBM Cloud [55].

- *Private Cloud*: the cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises. Offered cloud services are internal to the organization owning the infrastructure. Besides physical machines, a private cloud solution typically comprises an infrastructure management software solution that enables organizations to deploy resources effectively, monitor the health and performance of the infrastructure and eases the administrative tasks involved in managing large scale infrastructures. Examples of private cloud providers are OpenNebula [184] and OpenStack [185].

- *Hybrid Cloud*: the cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community or public) that remain unique entities, but are bound together by standardized or proprietary technologies that enable data and application portability. The public and private clouds in a hybrid cloud arrangement are distinct and independent elements. This allows organizations to store protected or privileged data on a private cloud, while retaining the ability to leverage computational resources from the public cloud to run applications that rely on this data. This minimises the data exposure, as sensitive data is stored only on the private cloud component.

- *Federation (or Community) Cloud*: the cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g. mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise.

1.3.3 Cloud Actors

According to the NIST Cloud Computing Reference Architecture [151] there are five major actors involved in the Cloud Environment: cloud consumer, cloud provider, cloud carrier, cloud auditor and cloud broker. Each actor is an entity (a person or an organization) that participates in a transaction or process and/or performs tasks in the cloud computing environment.

- *Cloud Consumer*: is the principal stakeholder for the cloud computing service. He represents a person or organization that maintains a business relationship with, and uses service from, Cloud Providers. The Cloud Consumer requests the required service from the provider's

service catalog, sets up the service contracts with the cloud provider and starts using the cloud service. Cloud consumers need SLAs to specify the technical performance requirements fulfilled by a cloud provider. SLAs can cover terms regarding the quality of service, security, remedies for system failures. A cloud provider may also list in the SLAs a set of promises explicitly not made to consumers, i.e. limitations, and obligations that cloud consumers must accept.

- *Cloud Provider*: is a person, organisation, or entity responsible for making a service available to interested parties. A Cloud Provider acquires and manages the computing infrastructure required for providing the services, runs the cloud software that provides the services, and makes arrangement to deliver the cloud services to the Cloud Consumers through network access. The cloud provider conducts its activities in the areas of service deployment, service orchestration, cloud service management, security and privacy.

- *Cloud Carrier*: is an intermediary that provides connectivity and transport of cloud services from Cloud Providers to Cloud Consumers. Cloud carriers provide access to consumers through network, telecommunication and other access devices (e.g. computers, laptops, mobile phones). The distribution of cloud services is normally provided by network and telecommunication carriers or a transport agent, where a transport agent refers to a business organization that provides physical transport of storage media such as high-capacity hard drives. The cloud provider will set up SLAs with the cloud carrier to provide services consistent with the level of SLAs offered to cloud consumers, and may require the cloud carrier to provide dedicated and secure connections between cloud consumers and cloud providers.

- *Cloud Auditor*: is a party that can perform an independent examination of cloud service with the intent to express an opinion in terms of security controls, privacy impact, performance, etc. Audits are performed to verify conformance to standards through review of objective evidence.

- *Cloud Broker*: is an entity that manages the use, performance and delivery of cloud services and negotiates relationships between cloud providers and cloud consumers. As cloud computing evolves, the integration of cloud services can be too complex for cloud consumers to manage. A cloud consumer may request cloud services from a cloud broker, instead of contacting a cloud provider directly.

Chapter 2

Assessment of Dependable Systems

In the process of designing and implementing solutions to improve the dependability of a system, an important key point regards the evaluation of performances of the target system under realistic conditions. Specifically, modern systems have to be designed in order to cope with variable workload, and thus the assessment methodology plays a paramount role. For this scope, two important activities are (i) the workload generation towards the target system, and (ii) a proper benchmarking of the target system. There is no standard methodology for conducting these activities, and in this research area various solutions have been proposed, depending on specific experimental goals and needs.

2.1 Workload Traces

Workloads to be generated against a target system can be either synthetic or a reproduction of a real execution trace.

- *Synthetic Workloads*: are artificially created with specific programs such that they reflect typical workload patterns of the target system. They are best suited to test the behaviour of the target system and/or of its component under certain conditions or in specific scenarios. For example, it is possible to create a synthetic workload trace with sudden increase in workload that causes a corresponding sudden increase in resource demand, and test the performance of the target system in this specific scenario, or even assess the efficiency of the system when such a situation happens. However, it may be very hard or even impossible to artificially create a synthetic workload that reflects a real workload to which the target system may be exposed in real-world scenarios. A number of work and framework are adopted for this scope as [25] in which the authors proposed the Georgia Tech Cloud Workload Specification Language (GT-CWSL), that provides a structured way for specification of application workloads.
- *Real Workload Traces*: are obtained from real application executions by recording a particular metric over time (e.g. submitted requests in the last minute). The execution of a real work-

load trace on the target system can significantly help to evaluate its performance in a real-life scenario and possibly to assess the effectiveness of the auto-scaler intended to be integrated into the target system. However, there are very few publicly available real traces that can be used to generate a proper real workload. The Internet Traffic Archive [17] collects a number of different workload traces; the most largely used traces are from Internet servers, such as the ClarkNet trace [86] or the World Cup 98 trace [87]. They both contain the HTTP requests received by respectively the ClarkNet server over a two-weeks period in 1995, and by the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. Although such traces have been used in literature to assess performance of a wide variety of target system, they can only be properly used to evaluate performance of web-based applications. Each type of application (e.g. a web service, or an application service, a distributed storage system) may have particular unique patterns in the workload trace, thus a proper evaluation of such different application types should be conducted using the adequate real workload trace. Unfortunately, apart from the aforementioned real traces for web based application, there are just very few other options, especially for cloud-based applications.

2.2 Workload Generator

A fundamental component in the performance evaluation of a complex target system, and possibly of its associated auto-scaler, is the workload generation engine, which is in charge of reproducing the specified workload trace in the most accurate and efficient way. Modern distributed systems are characterized by a huge number of user requests, and a proper workload generation activity should be conducted such that the quantity of offered workload is compatible with workload in real life scenarios.

2.2.1 Workload Generator Engines

In the following are illustrated main characteristics of Workload Generation Engines:

- *Architecture*: in a centralized architecture, a single instance of the workload generator runs on a single node, whereas in a distributed architecture the engine is spread across multiple nodes. A centralized architecture is easier to setup and does not require any coordination mechanism. However the amount of workload that can be generated from a single node may not be sufficient to identify performance characteristics or bottlenecks in the target system. A distributed workload generator in turn provides capabilities to distribute the required amount of workload among multiple generator nodes in order to reach adequate levels of workload intensity. However, a distributed architecture requires a coordination schema in order to coordinate the workload execution among various generator nodes. Typically, a master-slaves architecture is employed, where a master node does not actually generate

workload but is in charge of distribute workload specification among slave nodes and synchronize their generation activities.

- *Performance metrics collection*: the workload generator engine may optionally provide the ability to properly collect required performance metrics from the target system. Collection and storing of such metrics should be conducted with a customizable granularity and such that there is no impact on performance of the target system.
- *Workload trace type*: the workload generator should provide the ability to easily create and execute synthetic and/or real workload traces.
- *Ability to interact with the target system of interest*: the majority of currently available workload generators provide the possibility to interact only with web-based systems by generating various types of HTTP requests towards the target system. Only few available tools permit to generate load against the wide range of available distributed storage systems.

2.2.2 Workload Generator Tools

Examples of currently available workload generators, along with their main features, are the following.

- **Apache JMeter** [130]: JMeter is a Java open source workload generator used for load testing and performance analysis. It can be used to test performance of various systems like Web-services, Databases, FTP Servers and other resources. It can generate heavy loads for a server, network or object, either to test its strength or to analyse overall performance under different scenarios. It has a full multi-thread engine that allows concurrent sampling by many threads, that may be grouped into separate thread groups each one of them emulating a different type of requests. JMeter provides a GUI for facilitating the configuration of performance tests as well as the analysis of test results. Furthermore, it provides a documented APIs for extensibility purposes and development of customized plugins in order to let JMeter interact with the desired target system. It can be deployed both in a single node architecture, or in a distributed master-slaves architecture when the amount of load to be generates is high enough to require more than one generator node. It provides the ability to collect metrics from the system under test. Numerous plugin available can help to generate synthetic workload traces, but those that provide the possibility to generate real workload traces have performance problems.
- **Rain** [29]: Rain is a statistics-based workload generation toolkit that uses parameterized and empirical distributions to model the different classes of workload variations. Rain provides a set of Generator APIs in order to be easily extensible via user-defined load generators targeting new systems and applications. Its architecture supports multiple workload generation strategies (open loop, closed loop and partly-open loop). It is intended to be well

suited for cloud workload generation. Unfortunately, documentation is minimal and lacks detail.

- **Httperf** [171]: Httperf is a tool for measuring web server performance. It provides a flexible and easy generation of various HTTP workloads and measuring server performance. Synthetic workloads are suitable to carry out controlled experimentation. It is possible to tune the workload in order to test the system under different number of users or request rates, with smooth increments or sudden peaks. However, it is specifically tailored for web-based target systems, it does not support a distributed architecture and it does not provide the possibility to generate workload based on a real workload trace.

- **Faban** [94]: Faban is a Markov-chain-based workload generator, and is widely used for server performance and load testing. It provides the possibility to collect performance metrics, and automate statistics collection and reporting. Faban supports numerous servers such as Apache httpd, Sun Java System Web, Portal and Mail Servers, Oracle RDBMS, memcached. Faban permits to build and modify realistic workloads traces starting from a log file. Due to its distributed and scalable architecture, Faban is well suited for generating cloud computing workloads. However, it does not provide the possibility to easily develop and integrate plug-ins in order to interact with other target systems.

- **Tsung** [152]: Tsung is a distributed load testing tool. It is protocol-independent and can currently be used to stress HTTP, SOAP, PostgreSQL, MySQL, LDAP and Jabber/XMPP servers. It can be deployed on single node architecture, and in a distributed architecture as well. Main feature is its ability to simulate a large number of simultaneous users even with single generator node. When used in a distributed mode, it is possible to generate a huge workload intensity even with a modest generator cluster. Its high performance are achievable due to its implementation in Erlang [14], a concurrency-oriented programming language, that made possible to support hundred thousands of lightweight processes in a single virtual machine. Tsung may be deployed on a cloud like Amazon EC2. However, it does not provide the possibility to easily develop and integrate plug-ins in order to interact with other target systems.

2.3 Benchmarking Tools

Application benchmarks are used to evaluate performance and scalability of a target system. Typically, they comprise a web application together with a workload generator that creates artificial session-based requests to the application. Overview of available tools. Examples of currently available benchmarking tools, along with their main features are the following:

- **RUBiS** [210] is a prototype of an auction website modeled after eBay.com. It offers the core functionality of an auction site (selling, browsing and bidding) and supports three kinds of

user sessions: visitor, buyer, and seller. The tool basically comprises a three-tier web application, where the three layers are implemented by an Apache web server with load balancing capabilities, a JBoss application server and MySQL database server. The last update of the RUBiS project was in 2008, but it is still a valid solution when it is needed a complete web-based multi-tier application to be used in testing and benchmarking activities. However, very basic workload generation and performance collection capabilities are provided.

- **TPC-W** [164] is a web server and database performance benchmark, proposed by Transaction Processing Performance Council (TPC), a non-profit organization founded to define transaction processing and database benchmarks [61]. This benchmark defines a complete Web-based book shop application where it is possible to simulate three different interactions: searching, browsing and ordering books. The system under testing needs to provide the implementation of this shop. The performance metric reported is the number of web interactions processed per second. It was declared obsolete in 2005. Although, it is still being used by the research community.

- **CloudStone** [212] is a multi-platform, multi-language performance measurement tool for Web 2.0 and Cloud Computing, developed by the Rad Lab group at the University of Berkeley. It is intended to be deployed on Amazon Elastic Cloud Computing (EC2), and primarily used to measure database performance. CloudStone uses Faban workload generator to generate load against a realistic Web 2.0 application called Olio, a two-tier social networking benchmark, with a web frontend and a database backend. The application metric is the number of active users of the social networking application, which drives the throughput or the number of operations per second.

- **VMmark** [158] is a benchmark tool suite used to measure performance, scalability, and power consumption of applications running in virtualised environments. The suite measures the performance of virtualised servers while running under load on a set of physical hardware. In order to measure the efficiency of the virtualisation layer, the suite runs several virtual machines (VMs) simultaneously, each of them configured according to a template that mimics typical software applications found in corporate data centers. The default templates are provided: email servers, database servers, and Web servers. The VMmark software collects performance statistics that are relevant to each type of application, such as commits per second for database servers, or page accesses per second for web servers. VMs are grouped into logical units called "tiles". When evaluating the system's performance, VMmark first calculates a score for each tile, based on performance statistics produced by each VM, and aggregates the per-tile scores into a final number.

- **YCSB** (Yahoo! Cloud Serving Benchmark) [58] is an open source framework for evaluating and comparing the performance of multiple types of data-storage systems, including NoSQL stores such as Apache HBase, Apache Cassandra, Redis, MongoDB, and Voldemort. The

common and most adequate use of the tool is to benchmark multiple systems and compare them. The framework is composed by two main components: the YCSB Client, an extensible workload generator, and the Core workloads, a set of common workload scenarios to be executed by the generator for evaluating the performance of different data stores. Workloads are defined in terms of percentage of insert, read, scan and update operations. Both components are extensible in order to define new and different workloads to evaluate system aspects or scenarios not adequately covered by the core workload, and to support the benchmarking of different databases. Main performance metrics reported are throughput and latencies. Although widely used to compare performance of different systems, YCSB workload generator component has very limited customization capabilities and permits to generate load only in terms of a given number of operations that have to be executed in a given amount of time, or at a given fixed rate. There are no means to generate workload according to neither a synthetic nor a real workload trace.

- **CloudSim** [43] is a Java framework for modeling cloud infrastructures and simulate the behaviour of deployed services under variable workloads. It has been developed by Cloud Computing and Distributed Systems (CLOUDS) Laboratory of University of Melbourne, Australia, and it is now considered the most popular open source cloud simulators. Initially, it was developed as a standalone framework; consequently, it has been defined further extensions as CloudReports [203] for the GUI, CloudSimEx [56] for a MapReduce and parallel computing simulations and Cloud2Sim [134] to run the framework among distributed servers, by exploiting the Hazelcast [146] distributed execution framework.

Part I

Autoscaling Techniques

Chapter 3

MYSE: An Architecture for Automatic Scaling of Replicated Services

In the first part of this thesis, we investigate solutions for automatic scaling a distributed system. The motivation for this study relies on an urgent need to balance performances and costs in private datacenter or Cloud environments. Indeed, nowadays Cloud Computing is fostering the adoption of distributed services. Such deployments allow to achieve high levels of availability and fast response times despite the heaviness of input loads. Industries which are owner of big Data Centers and Cloud XaaS providers aim to minimise the costs, while ensuring a desired performance level. In some case, a low level of performance may either bring additional costs or penalties, as in the case of a Cloud provider which violates a SLA with a client who stipulated a contract with specified QoS constraints. All of these variables make difficult to find runtime a configuration without waste resources.

Auto-scaling solutions have been widely proposed in literature [154] and allow to choose runtime the right computational resources (or configuration) to cope with fluctuating workload. However, generalising a definitive solution is challenging as different systems may require diverse approaches.

The timeliness in reacting to load variations plays a key role, as reconfiguration delays may either lead additional costs or threaten the system dependability. Indeed, if the system is not enough performant (due to under-provisioning periods) it can violate SLAs, or even worse, becomes unavailable. On the contrary, an over-provisioning approach might waste unused resources.

An additional challenge of designing elastic solutions is the cost of elasticity, i.e. the cost the service provider pays to activate/deactivate a replica like, for example, the bandwidth used to transfer the state from an active replica to a new one, the energy used to keep replicas running with low utilisation and the tradeoff between buying the infrastructure or just renting it. Usually, such costs occur each time that the system moves from one configuration to another and they may grow up if the *flipping phenomenon* (i.e. a sequence of activation and deactivation of replicas) is not properly mastered.

In this chapter, we propose a solution aimed at facing these issues. We claim that a proactive system may outperform a reactive one. Therefore, we designed the *Make Your Service Elastic* framework (MYSE) for the automatic horizontal scaling of a replicated service. By monitoring input requests patterns and the service times delivered by the replicated service, MYSE learns over time through Artificial Neural Networks (ANN)[1] how input load and service times vary, and produces estimations to enable early decisions about reconfiguration. A queuing model of the replicated service is used to compute the expected response time given the current configuration (number of replicas) and the distributions of both input requests and service times. A novel graph-based heuristic called *Flipping-reducing Scaling Heuristic* is employed that leverages this model to find the minimum number of replicas required to achieve the target performance and to reduce as much as possible the flipping phenomenon.

We carried out simulations by using a real dataset containing the requests to a website over the time. The results showed high accuracy in input traffic prediction and good effectiveness in taking proper scaling decisions.

Contributions. The content of this chapter has been partially published in [9]. The novelty of MYSE mainly consists in:

- combining together traffic forecasting (with artificial neural networks), and performance estimation through queuing models;
- addressing the problem of flipping by employing the innovative Flipping-reducing Scaling Heuristic;
- an extended model to cope with heterogeneous servers.

Chapter Structure. The chapter is organized as follows: related works are presented in Section 3.1; Section 3.2 presents the system model and the problem statement; Section 3.3 describes the MYSE architecture and the autoscaling solution; Section 3.4 reports the preliminary results obtained from simulations; Section 3.5 presents a revised solution to extend MYSE in a heterogeneous environment and Section 3.6 summarise the work and how to going along directions for real-world instantiations.

3.1 Related Works

According to a recent survey [154], there are several works on automatic scaling of elastic applications in the cloud. This survey proposes the following classification of the auto-scaling techniques existing in literature, on the basis of the approach they employ.

[1] see Appendix A for more details.

- *Static, threshold-based policies.* The configuration is changed according to a set of *rules*, some for scaling out and others for scaling in ([72] [105] [107] [161]). This is a completely *reactive* approach that is currently used by most of the cloud providers.

- *Reinforcement learning.* It is an automatic decision-making technique to learn online the performance model of the target system without any a priori knowledge. Such continuous learning is used to choose the best scaling decision according to the goals of decreasing response time and saving resources ([28] [73] [192] [220]).

- *Queuing theory.* The target system is modeled using techniques coming from the queuing theory with the aim of estimating its performance given a small set of parameters, like input rate and service time, that can be monitored at runtime ([220] [223] [226] [240]).

- *Control theory.* It is used to automate the management of scaling decisions through the employment of a feedback or feed-forward controller module whose objective is to meet performance requirements by adjusting the configuration of the target system ([34] [4] [186] [188] [236]). Feedback controllers correct their behaviour by taking into account the error reported by the target system through a *gain parameter* that can be adapted dynamically. Feed-forward controllers are based on model predictive control (MPC) and aim at forecasting the future behavior of the system. The relationship between the input (the workload) and the output (the configuration to adopt) is embedded into the transfer function, which can be implemented in several ways (i.e., smoothing spline, Kalman filter, Fuzzy model).

- *Time-series analysis.* It can be used to spot recurring patterns of the workload over time, in order to forecast future workload so as to come to a scaling decision early. Several techniques can be used like averaging methods, regression and neural networks ([44] [45] [50] [98] [120] [126] [128] [166] [200] [208]).

The limitations of an approach based on static, threshold-based rules lies in its reactive nature: it only takes action after the recognition of a situation that requires scaling, and during the time needed for the scaling to complete either the system provides poor performance or resources are wasted. This problem can be addressed by using time-series analysis in order to forecast how the workload is likely to change over time so as to enable scaling decisions in advance and consequently avoid transient periods where the system is not properly configured.

The drawbacks of the techniques based on reinforcement learning are the excessive length of the training phase before reaching a point where it becomes effective, and the difficulty to adapt to workloads that change quickly.

Using control theory can actually be a valid choice, but choosing the right gain parameter is hard. In fact, considering a fixed gain parameter, its tuning is hard and cannot be adjusted at runtime; on the contrary, using an adaptive parameter, that is changed according to the workload, is likely to introduce flipping.

The employment of queuing models to estimate performance can turn out to be not reliable enough because the hard assumptions it requires could be not valid in a real scenario. Nevertheless, we chose to model the replicated service using a queue model because it doesn't require a long training as reinforcement learning does instead.

In addition to the works cited in the survey, also others employ a proactive approach for auto-scaling as we do, but none combines together traffic forecasting through artificial neural networks, and performance estimation through queuing models. Ghanbari et Al. [95] present an auto-scaling approach based on MPC that aims both to meet SLA and save resources by framing the problem as an MPC problem.

Moore et Al. [170] describe an elasticity framework composed by two controllers operating in a coordinated manner: one works reactively on the basis of static rules and the other uses a time-series forecaster (based on support vector machines) and two Naive Bayes models to predict both the workload and the target system performance.

For what concerns how the flipping phenomenon is dealt with in literature, the survey [154] reports that such an issue is addressed in some of the threshold-based works by setting two distinct thresholds: one for scaling out and another for scaling in, so as to have a "tolerance band" that can absorb part of the oscillations. The survey also advises to set so called *calm periods* during which scaling decisions are suspended. Our approach is based on the concept of calm periods, as suggested in the survey, but is more refined as the length of such period is adapted on the basis of the amount of flipping experienced so far.

3.2 System Model and Problem Statement

We refer to a *cluster* as a fixed set of available servers (i.e. virtual or physical machines) of cardinality N. We assume servers to be homogeneous, i.e. they have the same finite computational power and can be used interchangeably. We refer to servers as *nodes*. We consider a *distributed service*, consisting of a set of *service instances* deployed over a subset of available nodes. At most one service instance is deployed on each node.

We refer to *configuration* as the number s of service instances deployed, hence s corresponds also to the number of used nodes. We modeled the service as a queuing system with s server [91] and a FIFO scheduling policy. We assume, moreover, unlimited buffer for requests and unlimited population dimension.

Thus, we refer to the queuing systems with the Kendall's notation [136] in the form $A/B/s$, where A refers to the distribution of the *arrival rate*, B refers to the distribution of *service time* and s to the aforementioned deployed service instances within the configuration. Figure 3.1 shows the queuing system. Such modeling permits to manage the flow of requests with an analytical approach based on the Queueing Theory [5] and it allows to study the queue lengths and waiting times.

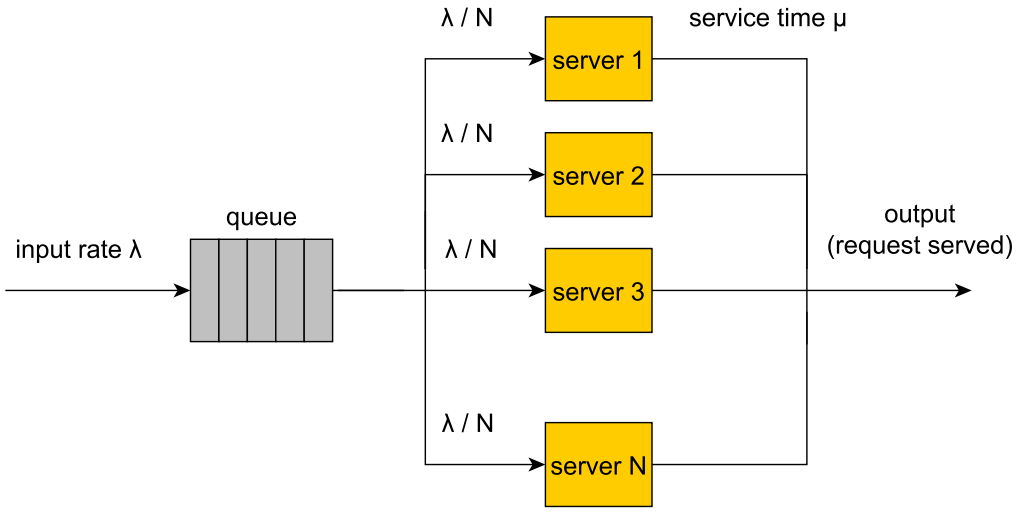


Figure 3.1: *Queuing System*

A number of clients can interact with the distributed service by sending *request* messages. We refer to *input rate* or *workload* $\lambda(t)$ as the number of requests per time unit issued by clients towards the distributed service at a given time t and we consider that the workload can change over time.

Once a request is received by the distributed service, a certain amount of time is required to serve it. We refer to that time as the *service time* characterised by a mean μ. We consider instead negligible the latency of the network, hence we can assume that the *response time* is equal to the service time plus the time for a request to pass from the queue into the service. The total amount of requests served by the distributed service per time unit is referred as the *throughput*.

According to an input workload, different configurations provide different performances. Specifically, we assume that a configuration with s servers has a lower response time than a configuration with $s - x$ servers, where $x = [1, s - 1]$, as having more servers it can drain the queue quickly. Indeed, considering $\rho = \lambda / s\mu$ being the utilisation factor representing the percentage of time the servers are busy, it is clear that by increasing s we obtain lower value of ρ.

Moreover, we work under the assumption that the system is stable; in queuing theory this condition is achieved when the system is in a *steady-state*, i.e. $\rho < 1$. This is a necessary condition to analytically estimate times and number of requests both in queue and in the global system [5].

We consider the workload to be dynamic, hence it can change over time according to specific pattern as we mentioned in Section 1.2.1. Thus, we consider that the configuration can vary over time by means of *scaling actions*: *scale-in* actions allow to reduce the configuration by removing one or more active service instances, while *scale-out* actions allow to augment the configuration by adding one or more service instances.

We consider that a scaling action takes a negligible time that we can consider fixed independently by the number of service instances added or removed. Besides, we assume the service to be stateless, and so, neither further actions are required to handle the state migration nor further time.

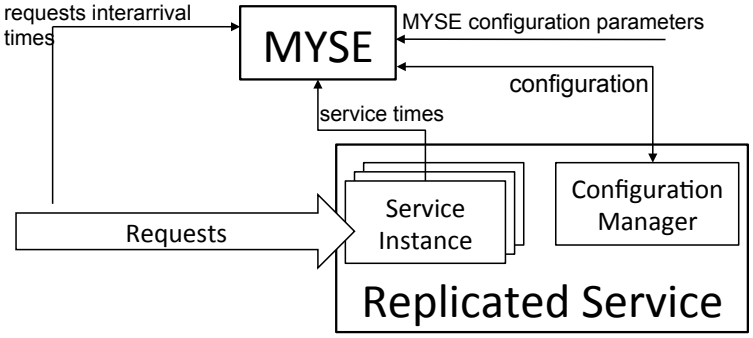


Figure 3.2: *Integration of MYSE module with target replicated service*

Considering a target desired performance level, i.e. a Quality-of-Service (QoS), as a maximum response time to ensure to clients, the problem to tackle in this work consists in dynamically find the minimum configuration able to achieve such a QoS.

3.3 MYSE Architecture

Figure 3.2 shows how the MYSE module is expected to be integrated with the target service. We assume that a Configuration Manager module is available to receive external commands that update the configuration.

The basic idea is to consider the replicated service as a black box and monitor requests patterns over time to identify the relevant characteristics of input traffic so as to properly reconfigure the service through the Configuration Manager. To this aim, we assume that replicated service instances export, as performance metrics, their service times. This allows us to follow the black box approach as in [200] by considering only observable parameters, like the requests patterns and the average request latency, without entering into the details of the specific service implementation and allows the MYSE module to work with several different types of applications.

Monitoring requests arrival and service times over time enables to predict their probability distribution. The queuing model is then used to compute the expected latency in serving a single request, which can be compared to given QoS requirements to figure out whether the compliance is actually achieved. The same queuing model is employed to derive the minimum configuration allowing to meet the QoS and, at the same time, to avoid wasting resources. We employed four ANNs[2], two ANNs to learn how request rate and request distribution vary over time, and other two to learn how service times and their distribution vary over time. In this way, we can conveniently update the configuration early enough to avoid temporary performance worsening or resource under-utilisation. The timeline of these predictions is provided externally (Δ parameter).

Figure 3.3 details the submodules of MYSE. The Δ-*Load and Distribution Forecaster* is in charge of learning and forecasting request rate and request distribution (it includes Δ-*Load Forecaster* and

[2]We consider that ANNs can be one of the best solutions for our requirements as they provide (i) a data-driven non-linear model and (ii) the ability to generalize and infer unseen parts of a population [239].

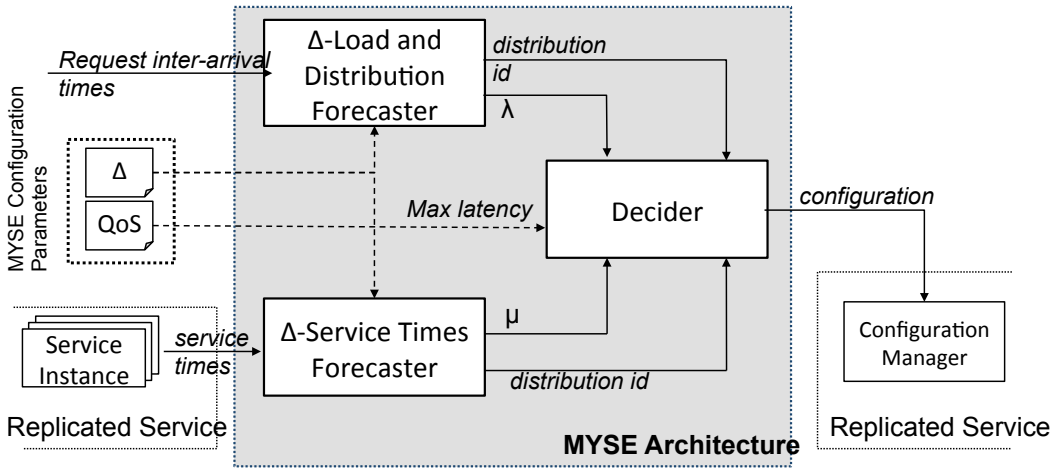


Figure 3.3: *MYSE Architecture*

Δ-*Distribution Forecaster* submodules). The Δ-*Service Times Forecaster* looks at service time patterns to extract the distribution of service times and its mean μ. The *Decider* determines the suitable configuration to meet QoS on the basis of the inputs supplied by the other two submodules and of the *Flipping Parameters* (see Section 3.3.4). Single submodules are described below.

3.3.1 Δ-Load Forecaster Submodule

It analyses request rates over time and employs an ANN to provide predictions on expected request rate λ within Δ time units.

Several guidelines are available for choosing the number of hidden layers and nodes of an ANN for obtaining good generalisation and low overfitting [239]. We followed these guidelines to empirically find the best ANN.

As it is considered that a single hidden layer is good enough to approximate any complex non-linear functions to any desired accuracy [239, 63], we designed a multivariate network similar to [76], composed by one hidden layer with 11 hidden neurons (that in our experimentation provided the best performances), with four input nodes for the date (day, day of the week, month, hour) and one input node for the last observed input load. The output node simply represents the predicted traffic values. Figure 3.4 shows the final architecture of the ANN used in the Δ-Load forecaster.

The ANN is trained runtime to update the timeseries of values to learn. The input dataset used for learning has a sliding window. We tuned the windows length in Section 3.4.

3.3.2 Δ-Distribution Forecaster Submodule

It is composed by two parts, the *Distribution Recogniser* and the *Distribution Forecaster*; it analyses requests inter-arrival times to produce predictions on request distribution Δ time units ahead.

Figure 3.5 shows the internal structure of the module. The Distribution Recogniser estimates the best-fitting continuous or discrete distribution by analysing a set of samples given in input,

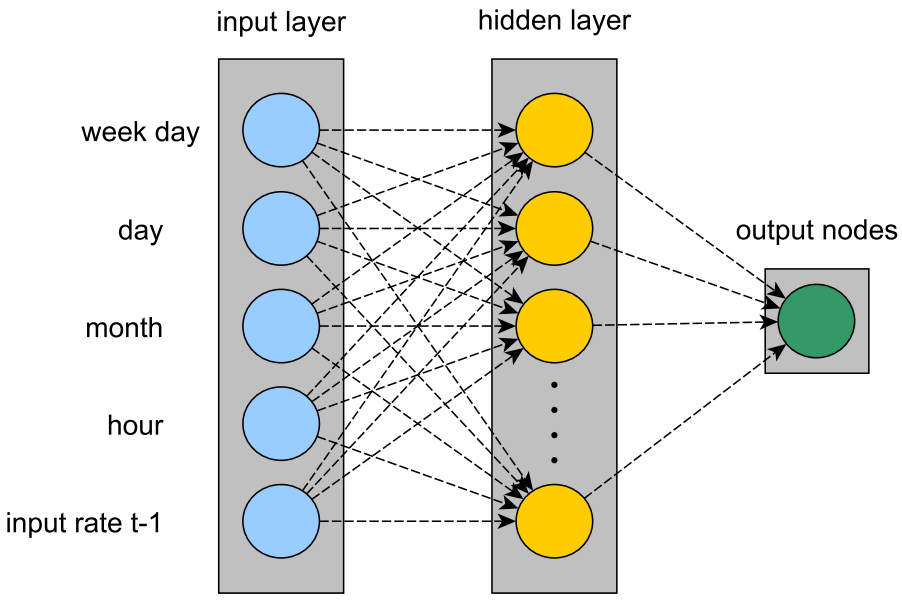


Figure 3.4: *ANN implementing the Δ-Load Forecaster submodule*

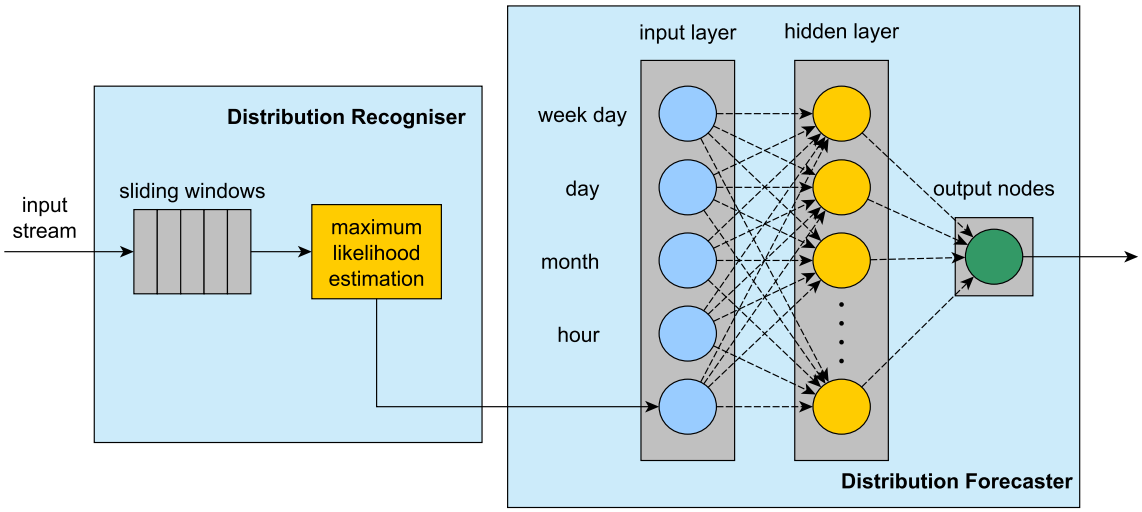


Figure 3.5: *Distribution Forecaster structure*

which represent the requests inter-arrival times. These samples are analysed with a fixed-length sliding window.

The estimation of distribution parameters is made by using the *maximum-likelihood estimation method* [173]. This result is then used to perform "goodness-of-fit" tests, such as the *chi-squared test* (for discrete distributions, i.e. Poisson) and *Kolmogorov-Smirnov test* (for continuous distributions, i.e. Normal), for discriminating among distinct distributions [32]. Both tests produce a "statistic significance level" *p-value* used to determine whether a group of samples comes from a specific distribution [194].

In the current implementation, the submodule is able to recognise and classify the following distributions: Poisson, Uniform and Exponential. This enables MYSE to employ several queuing models as *M/M/s*, *M/D/s*, *M/G/s*.

The Distribution Forecaster is able to predict the future distribution by using an ANN trained by taking as input the output produced by the Distribution Recogniser, which is an encoding of the recognised distribution. Such an encoding consists in assigning to each recognisable distribution a distinct numerical identifier, so that the identifiers are distanced enough to avoid possible conflicts. The choice to use two distinct parallel ANNs (this one and the one of the Δ-Load Forecaster) is made to improve the forecasting accuracy, as suggested in [239]. The ANN is built following the same empirical guidelines discussed in Section 3.3.1: it has as input four nodes for the date and one node for the previous distribution id, 150 hidden neurons and one output for the forecasted distribution. The learning rate is fixed to 0.9 and momentum to 0.4, while the other parameters are the same derived in Section 3.3.1.

3.3.3 Δ-Service Times Forecaster Submodule

It takes as input the service times provided by the replicated service and produces as output the estimation of their distribution and the mean μ of service times. The same techniques employed for the Δ-Load and Distribution Forecaster are used here so as to recognise the distribution of time series and to predict both their distribution and mean Δ time units ahead. Tracking only input/output traffic to derive the service times allows to be non intrusive with respect to target service, so, according to the black box approach, the service requests and responses can be traced by monitoring input and output requests.

3.3.4 Decider Submodule

The submodule computes the minimum configuration for guaranteeing QoS compliance in service provisioning (i.e. the latency in the specific case). It takes as input the latency threshold specified in the QoS, the predictions on request distribution (distribution id and λ), the distribution of service times (distribution id and μ) and the *Flipping parameters*, which are the tuning parameters of the Flipping-reducing Scaling Heuristic, described in detail later in this section. The Decider submodule first applies the well-known queuing model techniques to compute the expected latency T (i.e. the average waiting time of requests in the system) for the current configuration with s servers, given the request rate and the service times provided by the forecasters. For a *M/M/S* queue T is:

$$T = T^q + \frac{1}{\mu}$$

where T^q is the average waiting time of requests in the queue computed as:

$$T^q = \frac{N^q}{\lambda}$$

that depends from N^q, i.e. the average number of requests in the queue, computed as:

$$N^q = \frac{1}{s!}\left(\frac{\lambda}{\mu}\right)^s \frac{\rho}{(1-\rho)^2} p_0$$

depending from the probability p_0 that there are no customers in the queue, obtained as:

$$p_0 = \left(\sum_{n=0}^{s-1} \frac{1}{n!}\left(\frac{\lambda}{\mu}\right)^n + \frac{1}{s!}\left(\frac{\lambda}{\mu}\right)^s \frac{1}{1-\rho}\right)^{-1}$$

from which is finally possible to obtain the number of requests in the system, i.e. the number of requests in the queue N^q summed to the ratio of traffic intensity λ/μ:

$$N = \lambda T = N^q + \frac{\lambda}{\mu}$$

Then, it applies a heuristic to decide whether scaling out (in case QoS is violated), scaling in (in case a configuration with less replicas can still guarantee QoS compliance) or keeping the current configuration. We propose here a Simple Decision Heuristic and a Flipping-reducing Scaling Heuristic.

A Simple Decision Heuristic

The reference heuristic used for our tests is the simplest as possible and it is based only on finding the minimum number of servers to ensure the QoS wished. For doing that the Decider module employed several queueing models and chooses the one which is the best fit with the distribution given as input. The expected latencies are so calculated with formulas given by the model and hence the number of active servers in the new configuration are chosen. This heuristic does not consider previously QoS violation or flipping phenomena (consecutive switch-on and switch-off of a server), but permits us to evaluate the performance of MYSE with respect to the optimum number.

The Flipping-reducing Scaling Heuristic

If QoS compliance can be achieved with the current configuration (with s replicas), then the algorithm evaluates if switching off replicas still makes it possible to satisfy QoS requirements. It computes the maximum number n of replicas that can be removed without violating the QoS, and moves from a configuration with s servers to a configuration with $s - n$. On the contrary, if the expected latency is higher than QoS threshold, then the algorithm computes the minimum number n of replicas to activate in order to comply with the QoS, and moves from a configuration with s servers to a configuration with $s + n$. Highly variable traffic and prediction errors can make the configuration oscillate very often, introducing a lot of overhead due to frequent activation/deactivation of replicas. This phenomenon is referred to as *flipping*, and we dealt with it by introducing a *cost function* that prevents the algorithm from moving back in a certain configuration in case such configuration was set too recently.

To this aim, we defined an edge-weighted directed graph $G = (V, E, w(e,t))$ where (i) the set of vertex V represents all the possible configurations (i.e. number of active replicas) i.e. $V = \{1, 2, 3, \dots s_{max}\}$, (ii) there exists an edge between any pair of vertexes (iii) for any edge $e_{s,s'} \in E$ the edge weight $w(e_{s,s'}, t)$ represents the cost of moving from the configuration s to the configuration s' at the current time t.

The cost function is defined as $w(e_{s,s'}, t) = FlippingCost \cdot flipping\%$, where $FlippingCost$ is one of the Flipping parameters, while $flipping\%$ is computed as the number of the flippings detected from the beginning divided by the number of heuristic executions. In this way, the $flipping\%$ decreases in time. The detection of a flipping is triggered when two scaling decisions in opposite directions (i.e., a scaling out and a scaling in) are executed within a configurable window (the $FlippingWindow$, another Flipping Parameter). When the algorithm moves from the configuration s to the configuration s' at time t, the edge $e_{s,s'}$ is assigned with the value $w(e_{s,s'}, t)$. Then, such weight is decreased linearly in time until it comes back to 0. The transition from a configuration s to a configuration s' is allowed at time t only if $w(e_{s,s'}, t) = 0$.

3.4 Simulations

3.4.1 Environment

In this section we describe a set of simulations aimed at assessing the effectiveness of the proposed architecture. In particular, we first evaluated the ability of the Δ-Load Forecaster to forecast the workload and adapt to different types of loads, then we evaluated the Δ-Distribution Forecaster and finally we evaluated globally the goodness of the approach by evaluating the evolution of the configurations in time with both proposed scaling heuristics. In all these simulations, Δ is fixed to one hour, but we also show which is the best setting for such a parameter in our scenario. We didn't simulate the Service Times Estimator because of the unavailability of traces describing service times over time, but we considered it as fixed and known to the Decider Module.

As a reference dataset, we used a real trace with 8000 hours provided by Google Analytics framework on our department services to train and test the ANNs. It has been divided in three parts: 70% for training, 15% for validation and 15% for test using a k-fold cross validation with $k = 10$ in order to choose the right ANN parameters and address overfitting. We normalised the input parameters with the *Max-Min Normalisation*:

$$v' = \frac{v - \alpha_{min}}{\alpha_{max} - \alpha_{min}} * (\beta_{max} - \beta_{min}) + \beta_{min}$$

where v' is the original value v normalised, α_{min} and α_{max} are the old min and max value, while β_{min} and β_{max} are the new min and max value, in our case $[0, +1]$.

We employed the *Backpropagation Algorithm* [202] for learning, with a sigmoid as activation function. A general method to set the network parameters doesn't exist and the common practice

is to set the parameter empirically [239], so we fixed *learning rate* and *momentum* to 0.3 and 0.5, respectively, by executing several tests aimed at trading off the recognition error with the exposure to overfitting.

All simulations has been conducted on a machine equipped with an Intel Core 2 2.00 GHz and 2 GB RAM. For measuring the error rate committed in our forecasting we used mainly three metrics:

- Root Mean Square Error (RMSE): $RMSE = \sqrt{\frac{\sum_{i=0}^{n}(f_i - y_i)^2}{n}}$
- Mean Absolute Error (MAE): $MAE = 1/n * (\sum_{i=0}^{n} |f_i - y_i|)$

where f_i and y_i are, respectively, the normalised predicted value calculated with the forecasters and the normalised real value we had in our dataset.

3.4.2 Evaluation of the Δ-Load Forecaster

We carried out some simulations to check how long it takes for the ANN to be properly trained as the number of backpropagation iterations varies. The obtained results show that using up to 100 iterations allows to keep the training time short enough, i.e. below 1.2 seconds in our testbed.

In order to show the accuracy of prediction and the adaptation capability to changes in the load pattern, the reference dataset has been integrated with a synthetic one. In particular, after three days, we appended other five days of the same dataset that has been scaled (i.e. amplified by three times the real values), and a sine function oscillating between 0 and 200 requests per second.

To make the ANN adaptive to traffic changes, an online learning is employed as follows. We store the training set in a fixed-length sliding window containing the last 100 inputs, and at each new input the ANN is trained using all the inputs in the window by executing 100 iterations of the backpropagation algorithm. Figure 3.6 shows the comparison between the real number of requests over time and the predictions. As we can see, the predictions follow the real pattern and converge quite quickly after a request pattern changes. The weekly RMSE of predictions is 4%, and the MAE is 3%.

Furthermore, we evaluated the accuracy of prediction by comparing the forecasted values obtained with different horizons of prediction Δ. Figure 3.7 shows such results for $\Delta = 1, 2, \ldots, 8$ in term of MAE. It is possible to note how an horizon of prediction $\Delta = 2$ allow to achieve the minimum error.

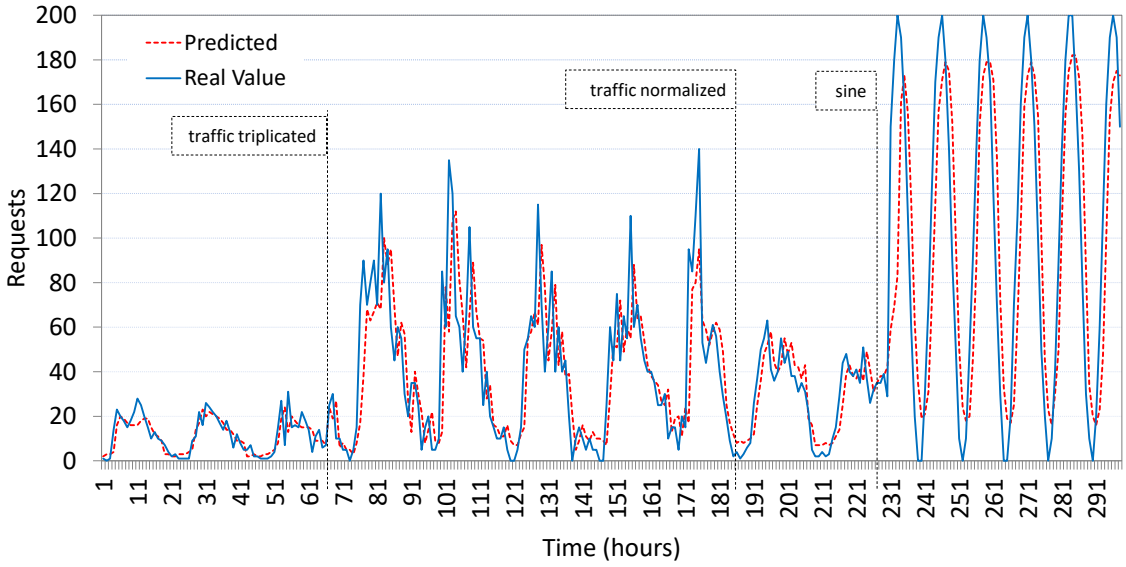


Figure 3.6: *Comparison between dataset traffic and forecaster predictions. The instants in time are indicated where traffic pattern changes: from normal to triplicated, to normal again and finally to a sine function with amplitude 100 is used.*

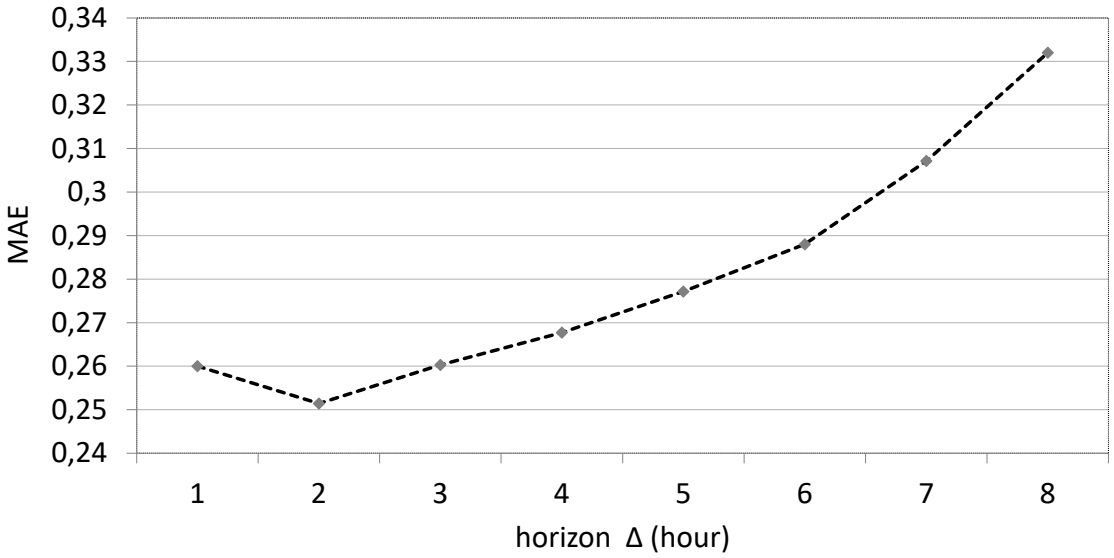


Figure 3.7: *Error (MAE) with different horizons of prediction*

3.4.3 Evaluation of the Δ-Distribution Forecaster

We evaluated the capability of this module to recognise a distribution by using a synthetic dataset where the distribution of inter-arrival times changes very often over time, from Uniform to Poisson and viceversa.

We measured the number of iterations required to correctly recognise distribution changes, where one iteration corresponds to the analysis of the samples of a single window. We computed

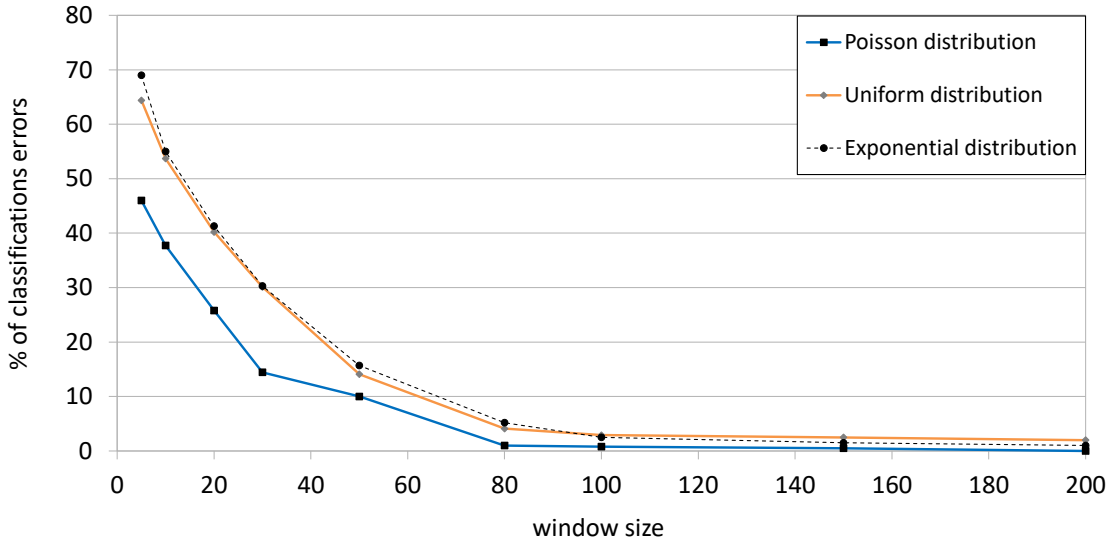


Figure 3.8: *Classification error in the Δ-Distribution Recognised while increasing the window size*

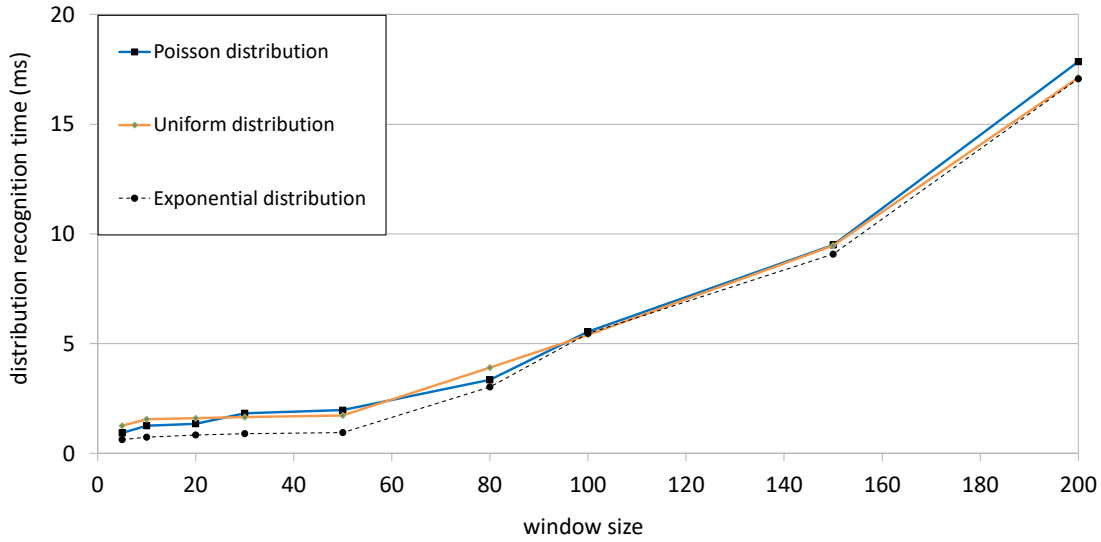


Figure 3.9: *Time needed by the Δ-Distribution Recognised to get the estimation of the distribution type by varying the window size*

that such a sliding window length has to be greater than 80 samples to reduce the estimation error below 5% (see Figure 3.8), and smaller than 100 samples so that the estimation latency is below 5 ms (see Figure 3.9).

We evaluated, then, the number of iterations needed to converge from a distribution to another. These results are shown in Figure 3.10, and the results showed how 62 iterations are required on average to detect the transition from Uniform to Poisson, while only 30 are needed instead for the opposite transition. Since the window slides at each sample, these results indicate that distribu-

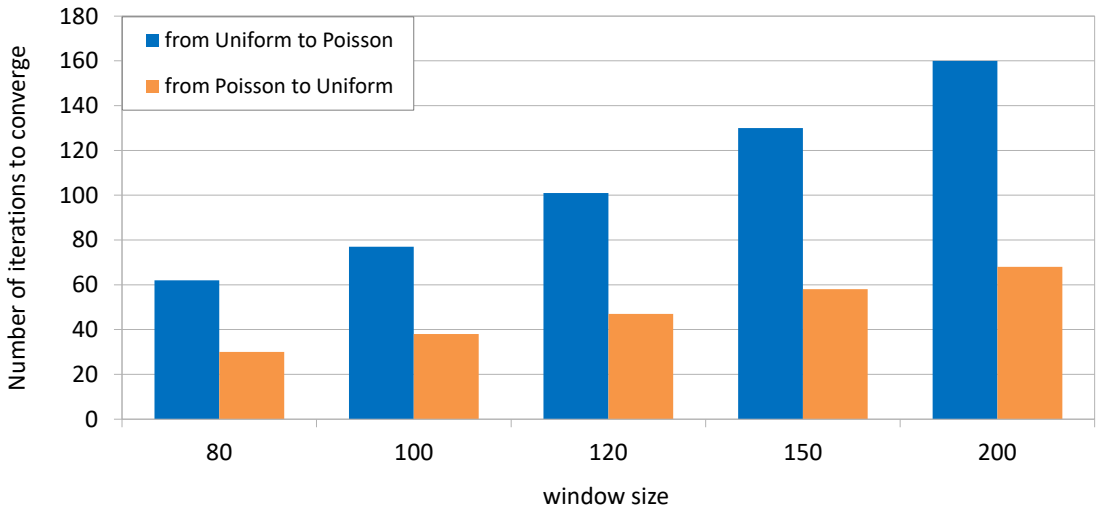


Figure 3.10: *Number of iteration needed to recognise a transition from a Poisson distribution to a Uniform and viceversa*

tion changes can be recognised before the window gets totally renewed. The reason why detecting Poisson to Uniform transition is much faster than the other lies in the ease of excluding Poisson when enough samples with zero deviation (that is, a Uniform distribution) are received.

We also evaluated the predictive accuracy of the ANN, by comparing real and forecasted request distributions over time. Three distinct distributions are recognised, each mapped to distinct ids chosen so that classification errors get minimised as shown in Table 3.1.

Distribution	**numerical** *id*
Poisson	0
Uniform	3
Exponential	11

Table 3.1: *Distribution Encoding*

We modified the dataset used for the evaluation of the Δ-Load Forecaster in such a way that the distribution of inter-arrival times changes very often over time among the recognisable distributions. Figure 3.11 shows that predictions are notably accurate with values of RMSE and MAE equal respectively to 4% and 3% over 3 days.

3.4.4 Evaluation of the Overall Architecture

This evaluation is aimed to highlight the relevant added value of employing traffic predictions for correctly issuing resource provisioning. The dataset used here is the Google Analytics one.

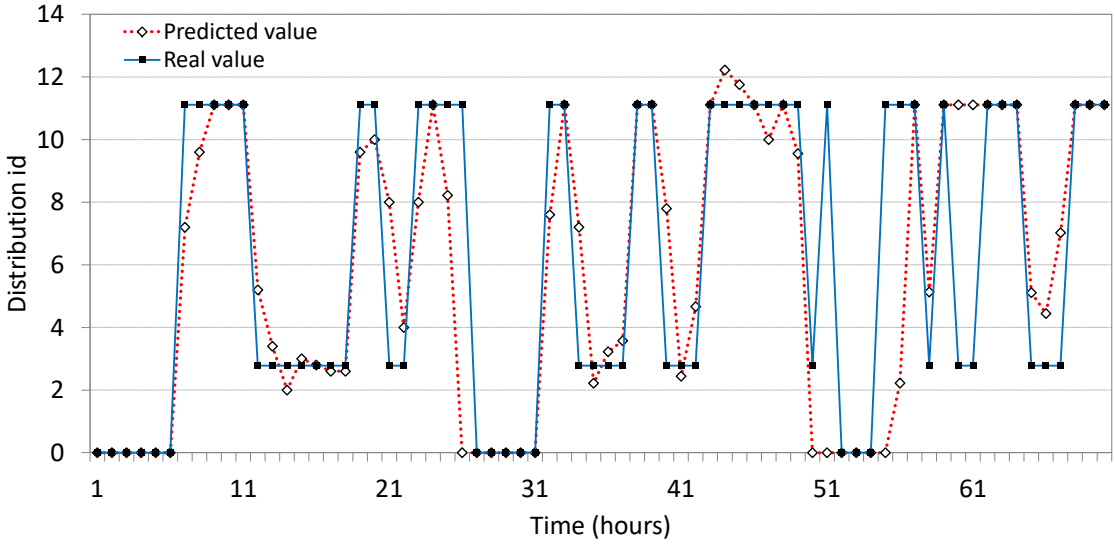


Figure 3.11: *Comparison between real and predicted request arrival distributions. Three distributions are recognisable: Uniform (id=0), Exponential (id=3) and Poisson (id=11).*

Figure 3.12 shows how the number of requested replicas varies over time on a hour-basis during a whole day. These results have been obtained by simulating three distinct scenarios: (i) the optimal configuration, that is the minimum number of replicas required to meet QoS requirements, (ii) the configuration produced without the contributions of Δ-Load and Distribution Forecaster, referred to as *Non-Trained MYSE* (NT-MYSE) and (iii) the configuration requested by the complete MYSE module. In the simulation of NT-MYSE, the Decider is fed with traffic details that are produced in real-time by the Distribution Recogniser on the basis of the current traffic only.

Until 8:00 the traffic is stable and both the approaches behave correctly, then traffic begins changing and the use of predictions shows its effectiveness by contributing to generate configurations that are nearer to the optimum compared to NT-MYSE approach. Another important advantage of employing predictions is rendering the system more robust to unexpected peaks. This can be seen by observing the effect of the isolated peak occurring at 16:00 (a peak in the number of the optimal number of replicas corresponds to a peak in traffic load): NT-MYSE is biased by such occurrence and hereafter keeps to over-provision, while MYSE correctly recognises it as an outlier. We quantified numerically the error of each approach by averaging over the entire day the number of replicas above (over-provisioning) and below (QoS violation) the optimum. NT-MYSE provided on average 0.29 replicas in excess and 0.33 replicas less than the minimum required, for a total of 0.62. MYSE allows on average to over-provision 0.13 replicas (55% more accurate) and under-provision 0.21 replicas (36% more accurate), that is a total error of 0.34 replicas (45% more accurate).

We evaluated, besides, the effectiveness in avoiding the flipping phenomenon by carrying

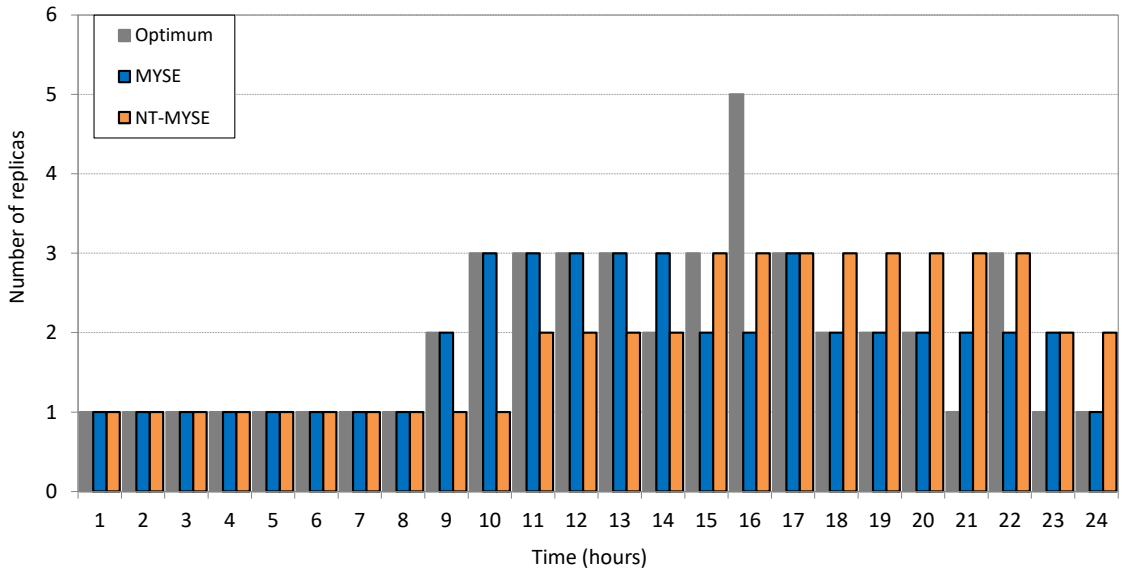


Figure 3.12: *Comparison between the number of replicas requested by MYSE and NT-MYSE, together with the indication of the optimum, that is the minimum number of replicas required to meet QoS requirements.*

out a simulation that compares the behaviour of the heuristic that uses the cost function with a heuristic that doesn't put any cost on the edges (i.e., $FlippingCost = 0$). We set $FlippingCost$ to 15000 and $FlippingWindow$ to 100 seconds. The results shown in Figure 3.13 give evidence that our heuristic is successful, indeed it manages to keep the the configuration fixed during the intervals when the other oscillates instead. It is to note that at the beginning the algorithm actually introduces flipping, but this is due to the fact the no flipping was occurred before, so $flipping\%$ is zero, yet.

3.5 System Refinement for Heterogeneous Servers

As our first system model for MYSE, many other work address the auto-scaling problem, by assuming homogeneous scenarios i.e. the resources have the same computational capabilities and can be used interchangeably [154].

In this section we proposed a revised model for heterogeneous machines which employs a solution based on *Q-Learning* (see Appendix B). Here we consider, so, VMs having different computational power and costs. Starting from no knowledge, the Q-Learning automatically learns the best policy to move from a configuration to another, given a requested workload. The goal is to find the closest configuration to the current one able to sustain the given workload and without waste resources. Two reward functions are defined and compared: (i) maximum reward is given if a state reached is able to ensure exactly the given workload and (ii) a reward that is function of the error committed between the given workload and the workload covered by the chosen configuration.

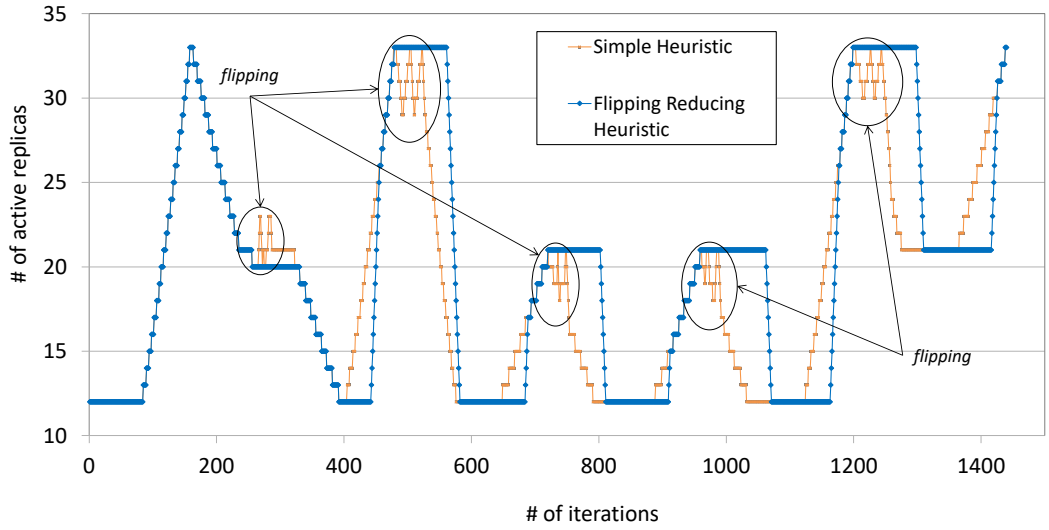


Figure 3.13: *A comparison between the heuristic employing a cost function and another one that doesn't use any cost function. Over time, it is shown that putting costs on the edges allows to limit the flipping phenomenon.*

The section is organized as follow: Section 3.5.1 introduce the system revised with the Q-learning approach to automatically scale heterogeneous resources, and Section 3.5.3 shows the simulations carried out and the results.

3.5.1 System Model

With reference to the original system model defined in 3.2, we consider to have a set of VM types $VM = \{m_1, m_2, \cdots, m_M\}$, each type i having a computational power cpu_i and a maximum number of available VMs max_i.

We define a configuration $config$ a set of VMs in which a type i could appear from 0 to max_i times. We define cpu_{config} the sum of all CPU values of VMs in the configuration.

We define a graph $G = (V, E)$ where $V = \{config_1, \cdots, config_N\}$ is the set of all possible configurations and E the set of edges that link two configurations. Note that an edge between two nodes (configurations) exists iff they differ for at most one VM and is bidirectional.

We assume that having the input rate is possible to estimate the computational power needed (i.e. the CPU) to handle such a workload.

3.5.2 Solution

We modeled the graph G for the Q-Learning algorithm as follow:

- *state:* a state $s \in S$ is a configuration i.e. a node in V;
- *action:* an action $a \in A$ is a move from a configuration to another i.e. an edge in E;

- *reward:* a reward is a number obtained by a reward function $R : S \times A \to \mathbb{R}$.

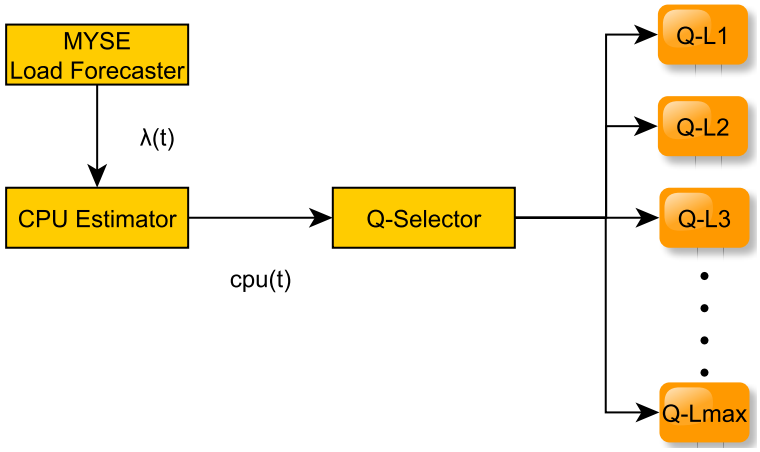


Figure 3.14: *Q-Learning System Architecture*

The solution relies on the MYSE Load Forecaster to predict the future input rate combined with a further module to estimate the CPU needed to handle such a workload. This module is the *CPU Estimator* and it replaces the queuing model employed for performance estimation in the original MYSE. Figure 3.14 shows the system architecture.

Besides, we have a set $\vec{Q} = \{Q_{cpu=1}, Q_{cpu=2}, \cdots, Q_{cpu=maxCpu}\}$ where each $Q_{cpu=j}$ contains the Q matrix related to computational power $cpu = j$, i.e. a matrix which provides the best action to choose to go from the current state to the goal state. At run-time, once $cpu(t)$ is estimated, a further *Q-Selector* module chooses the related matrix $Q \in \vec{Q}$ and it is used to select the aforementioned policy to select from the current state.

An initial phase of learning is necessary, so at beginning for each matrix $Q \in \vec{Q}$ the following procedure is executed: one state is selected as the goal state and iteratively, till a goal state is reached, from the current state s_t a random action a_t is selected, thus the Q-value $Q_{t+1}(s_t, a_t)$ is updated as follow:

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{opt future val}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

where t is the iteration, $Q_t(s_t, a_t)$ is the current stored Q-value by selecting the action a_t from the current state s_t, the learning rate α and the discount factor γ are two constant values, the value $\max_a Q_t(s_{t+1}, a)$ is the maximum Q-value obtained to go to s_{t+1} with an action a and finally R_{t+1} is a reward assigned according to a reward function. We defined two reward functions:

1. R_1 gives a reward only if the action a bring in a state with a configuration which has exactly the same cpu required, i.e. $cpu_{config} = cpu(t)$;

2. R_2 gives a reward that is function of the error committed between the cpu_{config} of the state reached and the requested $cpu(t)$.

Specifically, the equations 1 and 2 described the two Reward functions R_1 and R_2.

$$R_1(s,a) = \begin{cases} max\_reward & \text{if } cpu_i = cpu(t) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$R_2(s,a) = max\_reward - \frac{max\_reward}{penalty} * \underbrace{|cpu(t) - cpu_{config}|}_{error\ committed} \tag{2}$$

where

$$max\_reward = \underbrace{maxCpu - 1}_{maxerror} * penalisationFactor$$

and the *penalisationFactor*, used only in the reward function R_2, is a parameter grater than zero that the user gives to the algorithm to discourage the reaching of state that are not goal state, and $penalty = \sqrt{penalisationFactor}$.

3.5.3 Simulations

In this section we describe the simulation carried out and related results. we considered 4 type of VMs: M_1, M_2, M_3, M_4. The maximum instance number for each type M_i is set to 3 and their computational power $cpu_i = i$ just for simplicity. The resulted states with these parameters are 255 and each state has a configuration with the related computational power cpu_{config} in the range $[1;30]$. Figure 3.15 and Table 3.2 show a sketch of this topology. In the table bold values with a number of VM types grater than zero are bold.

For the simulations we used different *penalisationFactor* obtaining similar results, so we decided to report the result just for the value equal to 10. Similarly, we empirically evaluated different learning rate α and discount factor γ and here we reported simulations conducted with values set to, respectively, 0.1 and 0.9.

We compared the results of Q-Learning in terms of *step to goal* i.e. averagely how many step are necessary for each Q-Learning epoch to reach a goal state from any state. As it is possible to see from Figure 3.16 both the reward functions R_1 and R_2 have a similar behaviour, with initial *step to goal* high values that decreases exponentially in about 25 Q-Learning epochs to values between 2 and 6 for the two reward functions. After about 400 iterations the R_2 function cause an increasing value of the *step to goal* while R_1 continues to maintain the reached value. This is because with the R_2 function, we are assigning positives reward also to states that are not goal states, but having similar CPU values. In such cases, the best policy that Q-Learning chooses could not bring to a

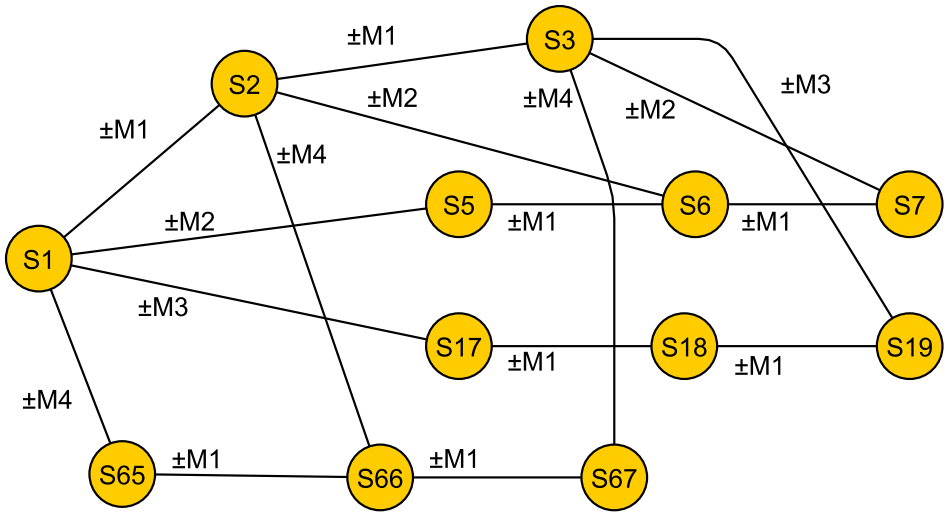


Figure 3.15: *Sketch of graph used for simulations*

goal state, preferring to reach near states in less steps.

Figure 3.17 shows the *minimum step to near goal state* by tolerating an error *err* for the reward function R_2. Thus it shows the minimum step necessary to reach an *acceptable state*, that is a state which differ at most of *err* CPU values from $cpu(t)$. It is possible to see how there are not so much differences in case of no error or few error and the global minimum (about 1.0 step) is reached with an error equal to 14 that is the $maxCpu/2$.

Obviously the reward function R_2 can bring to a state that is often not the goal state, but it is able to achieve such a state more quickly then R_1. Such differences depend from the connectivity of the graph, as in a connected graph, like the one used for this simulations, the difference between R_1 and R_2 can be lower than in a less connected graph. Furthermore, the tolerated error should be set according to a specific scenario. For instance, an alternative reward function may give different rewards according to a positive or negative error (i.e. over-provisioning or under-provisioning error).

3.6 Discussion

In this chapter we presented MYSE, an architecture for automatic scaling of replicated services with the goal of avoiding both performance worsening and over-provisioning. The proposed solution relies on the Queuing Theory so as to analytically estimate performances, whereas the input workload is forecasted through ANNs. As additional contribution we proposed a Flipping-reducing Heuristic based on a weighted graph, as an alternative to the typical *time guard* solutions [154], in which the scaling is suspended to avoid oscillations in the chosen configuration.

State	CPU	#M1	#M2	#M3	#M4
S1	**1**	**1**	0	0	0
S2	**2**	**2**	0	0	0
S3	**3**	**3**	0	0	0
S5	**3**	**1**	**1**	0	0
S6	**4**	**2**	**1**	0	0
S7	**5**	**3**	**1**	0	0
S17	**4**	**1**	0	1	0
S18	**5**	**2**	0	1	0
S19	**6**	**3**	0	**1**	0
S65	**5**	**1**	0	0	**1**
S66	**6**	**2**	0	0	**1**
S67	**7**	**3**	0	0	**1**

Table 3.2: *Sketch of states with the related amount of CPU and configuration*

Along this line, we proposed a solution for heterogeneous machines based on Q-Learning. Such an approach has the main objective to reduce the number of machines changed in a configuration so as to provide in two consecutive steps the most similar configuration able to sustain the new workload. This is motivated by the fact that a reconfiguration introduce an overhead which can lead either to an inefficiency resource usage or to a monetary cost as in the case of a public cloud with pay-as-you-go price model, where the user can be charged at every activation of a new machine.

By changing the Q-Learning reward function is moreover possible to find the most adequate tradeoff of a specific system between accuracy in performance provided and number of machine changed in two consecutive configurations.

This approach can be a pretty good solution, for instance, in a private cloud environment as it allows to reduce the number of machines changed in the configuration, hence the overhead. For a public Cloud where the customer can be charged on a fixed time unit (e.g. hourly), approaches that optimise resource saving and performances according to available resources in the current time unit can be assessed as alternative solution to avoid oscillations as future work directions.

According to the results obtained by the simulations carried out, we believe that (i) forecasting workload and service times and (ii) carefully modeling how performance gets affected, are the proper building blocks for achieving that objective. The main limitations of MYSE are (i) that it does not consider activation/deactivation times of new service instances and (ii) the analytical model based on the Queuing Theory cannot correctly model all real world application scenario.

In the next chapter we propose a refinement of the MYSE architecture to be more general and usable in real application scenario.

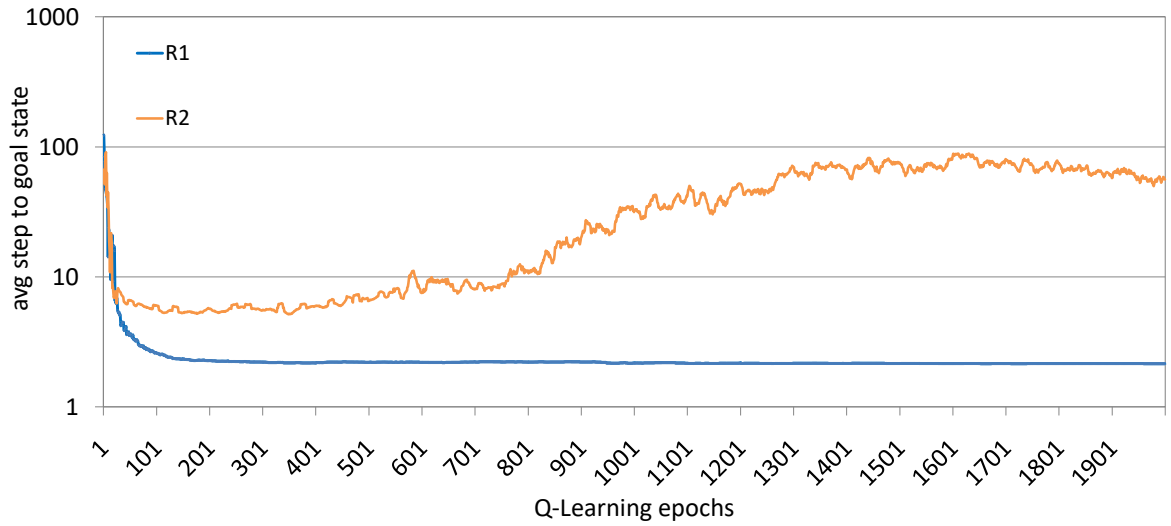


Figure 3.16: *Average step to reach a goal state from any state with the first 100 Q-Learning epochs*

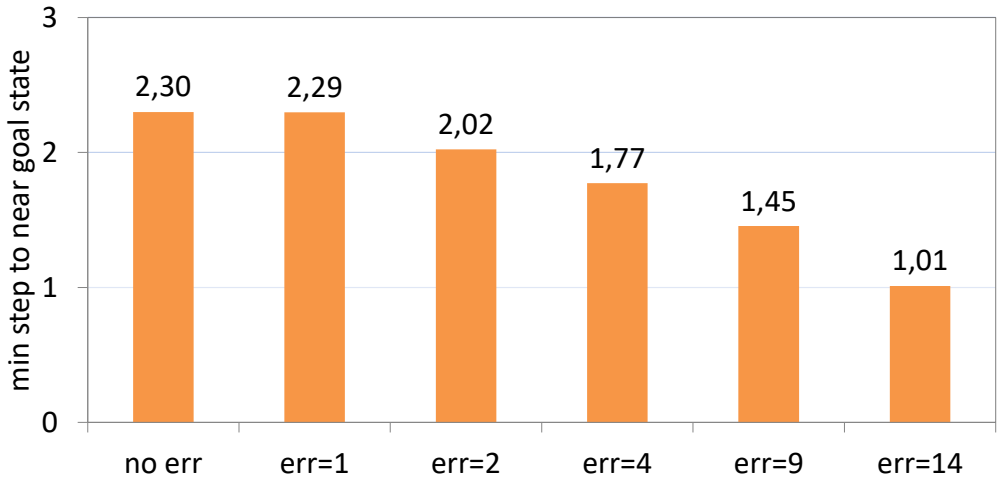


Figure 3.17: *Minimum number of necessary step to reach a near goal state*

Chapter 4

PASCAL: A Practical Proactive Autoscaling Architecture

In this chapter we investigate how to refine the MYSE architecture proposed in previous chapter to overcome its main limitations due to a simple model which does not consider transitory states after reconfigurations an due to the analytical approach which cannot model different real world scenarios. In literature, some requirements for an elastic controller of Cloud services have been proposed [150] and, specifically, they can be summarised as SASO properties [114]: *stability*[1], *accuracy*[2], *short settling time*[3] and *no overshoot*[4].

Here we present so PASCAL, an architecture for Proactive Auto-SCALing of generic distributed services. PASCAL instead of analytically predicting the response time of a given configuration, profiles the service to learn its behaviour and figure out the relationship between monitored metrics and performances to properly trigger scaling actions.

Chapter Structure. In Section 4.1 we present the system model refinement so as to propose in Section 4.2 the PASCAL architecture with the functional description of its internal modules and their matching with the modules of MYSE. Finally, Section 4.3 discusses the chapter and introduces the next two chapter introducing case studies for its employability.

4.1 System Model

We refer to a *cluster* as a fixed set of available servers (i.e.,virtual or physical machines) of cardinality N. Servers are homogeneous, they have the same finite computational power and can be used interchangeably. We refer to servers as *nodes*. We consider a *distributed service*, consisting of a set of *service instances* deployed over available nodes. At most one service instance is deployed on each node. Any service instance can be in one of these four states:

[1] *stability*: the system configuration does not oscillate.
[2] *accuracy*: the system configuration maximises the throughput.
[3] *short settling time*: the system quickly reaches a stable configuration.
[4] *no overshoot*: the system does not use more resource than necessary.

- *inactive*: the service instance is not running, thus it is not part of the distributed service;
- *active*: the service instance is running, thus it is part of the distributed service;
- *joining*: the service instance is becoming part of the distributed service but is not active yet;
- *decommissioning*: the service instance is leaving the distributed service but is not inactive yet.

We refer to *configuration* as the number of active service instances (i.e., service instances in the active state), which also corresponds to the number of used nodes. The configuration can change over time in response to *scaling actions*: *scale-in* actions reduce the configuration by taking active service instances to inactive state, while *scale-out* actions augment the configuration by making inactive service instances become active (see Figure 4.1). Such operations change even the number of active workers. Any state management for the distributed service, required to keep the service state across reconfigurations, is assumed to be handled by the service itself. A scaling action take a time T_{sa} which depends on the number of service instances to provide/remove: we refer to *joining_time* as the time needed for a service instance to switch from the inactive state to the active one during a scale-out action. Conversely, we refer to *decommissioning_time* as the time needed for a service instance to switch from the active state to the inactive one during a scale-in action.

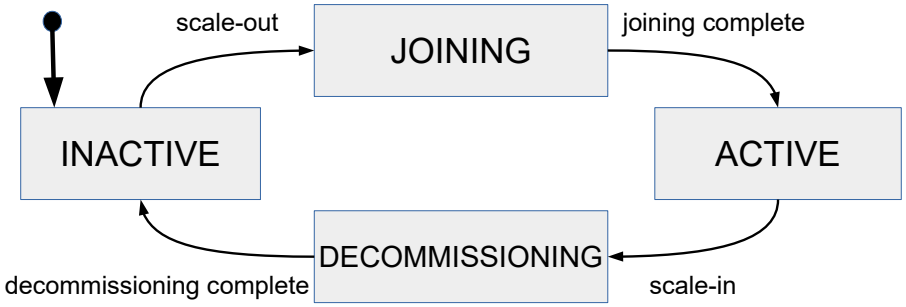


Figure 4.1: *State diagram of a service instance: in response to a* scale-out action, *it transitions to the* joining state, *which takes* joining_time *to complete and move to the* active state, *where the service instance can serve requests. A* scale-in action *brings an active service instance to* decommissioning state, *which takes* decommissioning_time *to complete and transition to* inactive state.

A scaling action *sa* is characterised by two points in time [139]: (i) *sa triggering point* (TP_{sa}), when the scaling action is triggered and the service instance transitions from inactive (active) to joining (decommissioning) state, and (ii) *sa reconfiguration point* (RP_{sa}), when the scaling action completes and the service instance passes from joining (decommissioning) to active (inactive) state. For a scaling action *sa*, joining/decommissioning time can be computed as the distance between TP_{sa} and RP_{sa}.

A number of clients interact with the distributed service by sending request messages. We refer to *input rate* or *workload* $\lambda(t)$ as the number of requests per time unit issued by clients towards the distributed service at a given time t. We consider also a temporal *horizon* h in which we can aggregate the workload, thus we define $\lambda_{max}(h)$ as the maximum workload for $t = t_0, \cdots, t_h$.

Once a request is received by the distributed service, a certain amount of time is required to serve it. We refer to that time as the *response time*, and to the number of requests served per time unit by the distributed service as the *throughput*.

Depending on the specific application scenario where the distributed service is employed, provided performances (i.e., response time and throughput) may need to ensure certain properties in spite of possible changes of the workload during time, e.g., response time should not exceed a given upper bound, or should not diverge over time.

A distributed service, setup with a given configuration, is said to *sustain* a certain workload if provided performances satisfy a given set of application-specific requirements. On the base of whether the current configuration is enough to sustain incoming workload, a distributed service can be in one of these two states:

- *normal*: the distributed service *sustains* current workload with the present configuration;
- *overloaded*: the distributed service *does not sustain* current workload with the present configuration.

If the distributed service is overloaded, there likely is some *bottleneck* to be solved through a scale-out action. Otherwise, maybe a scale-in action is required to decrease the number of used nodes and consequently save resources (e.g., save money by using less nodes).

A configuration is deemed *adequate* if it is able to handle the workload $\lambda_{max}(h)$ in the next horizon, hence if the distributed service is in normal state and we define *minimum configuration* as the smallest adequate configuration.

We denote as *sa demand point* (DP_{sa}) the point in time when a different configuration (i.e., the adequate configuration that should be setup by scaling action *sa*) is needed in response to some a workload.

The ideal situation to avoid performance worsening and maximise resource saving would be having $RP_{sa} = DP_{sa}$, i.e., the new configuration is ready exactly when it is required. In real settings, for a scaling action *sa* we have that one of these two situations occurs:

- $RP_{sa} > DP_{sa}$: the scaling action completes late, and the distributed service experiences a time interval of length $RP_{sa} - DP_{sa}$ where the configuration is still not adequate; in case of a scale-in action, this is a period of *over-provisioning* (since a smaller configuration could sustain the workload), while in case of a scale-out action this is an *under-provisioning* period (since there should be more active service instances);
- $RP_{sa} < DP_{sa}$: the scaling action finishes in advance, and the distributed service experiences a time interval of length $DP_{sa} - RP_{sa}$ where the configuration is still not adequate; in case of a scale-in action, this is a period of *under-provisioning* (since a larger configuration should be needed to sustain the workload), while in case of a scale-out action this is an *over-provisioning* period (since there should be less active service instances).

As scaling actions need some time to complete, the goal is identifying the minimum configuration early enough to minimise over-provisioning and under-provisioning periods (i.e., for the required scaling action sa, minimise $|DP_{sa} - RP_{sa}|$). Since we assume the workload is dynamic, the minimum configuration has to be computed periodically over time. We refer to this computation period as *configuration assessment period*. We assume that computing the minimum configuration takes much lower than the configuration assessment period, thus consecutive minimum configuration computations do not overlap.

4.2 PASCAL Architecture

PASCAL works in two consecutive phases: a profiling phase and an auto-scaling phase. The basic idea that underlines PASCAL is to use machine learning techniques to learn, during the *profiling phase*, the workload patterns (i.e., the *workload model*) and performance behaviours (i.e., the *performance model*) typical for the target distributed service. These models are then used at runtime, during the *auto-scaling phase*, to proactively scale the distributed service: future input rate is predicted on the base of the workload model, the corresponding minimum configuration is estimated using the performance model, and the consequent scaling action is triggered to minimise over/under-provisioning periods.

The architecture of PASCAL includes three main functional modules: (i) a Service Monitor collecting during both phases the metrics related to the target distributed service, (ii) a Service Profiler implementing the first phase and (iii) an AutoScaler for the second phase.

Figure 4.2 shows PASCAL's functional architecture and how it integrates with the target distributed service.

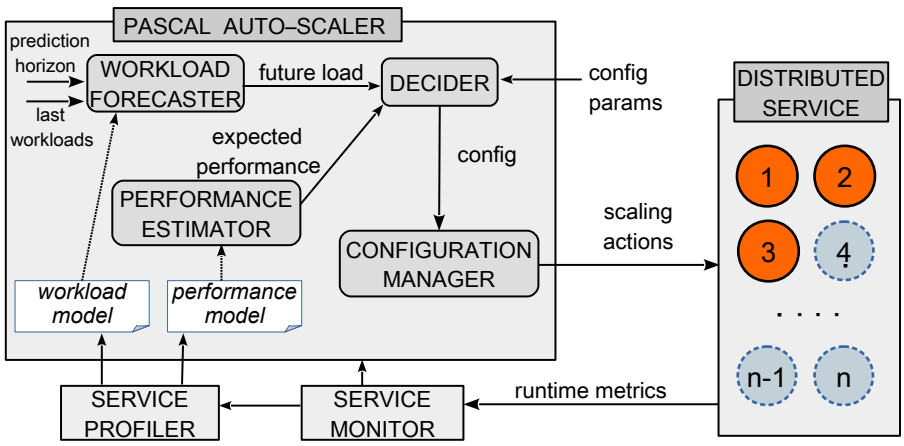


Figure 4.2: *PASCAL's functional architecture integrated with the target distributed service*

Next sub-sections detail these three main functional modules. The instantiation of this high-level architecture heavily depends on the peculiar characteristics of the distributed service to auto-scale. Anyway, its constituting modules capture the key aspects of realising a proactive auto-

scaling for a broad range of distributed services, thus PASCAL's architecture constitutes an effective guideline to instantiate specific solutions, as will be shown for the two application scenarios detailed in following two chapters, i.e. a distributed datastore and a distributed stream processing system.

4.2.1 Service Monitor

The *Service Monitor* monitors periodically at runtime a set of metrics related to service instances and machines hosting them. They are collected and used in both profiling and auto-scaling phases. Metrics can be either general (as CPU and memory) or specific of the service. A mandatory metric to collect is the input rate, which has to be used for both learning the workload model during the profiling phase and to predict future workload during the auto-scaling phase. According to the system requirements, to the chosen metrics and to the specific implementation, the monitoring can be implemented as more or less intrusive module.

4.2.2 Service Profiler and Performance/Workload Models

The *Service Profiler* is used during the profiling phase to learn the performance model and the workload model on the base of the metrics collected by the Service Monitor.

Different workloads are provided to the target distributed service for sufficiently long time, and related performance metrics are collected.

Collected metrics are processed to learn the *performance model*, which is used by the AutoScaler in auto-scaling phase to estimate the performance of the distributed service under a certain workload.

Real workloads in input to the target distributed service are observed over time, and related metrics are gathered. These metrics are analysed to learn the *workload model*, aimed at providing predictions as accurate as possible about what the workload will be like during the considered prediction horizon.

The duration of the profiling phase has to be tuned according to the specific scenario. The general idea is that the more data are collected the more the prediction will be accurate, but we cannot ensure this property as it is strictly related to the learning process [5].

4.2.3 AutoScaler

The AutoScaler comes into play in the auto-scaling phase. It periodically (i.e., once every *configuration assessment period*) uses its internal modules to (i) predict forthcoming input rate over the considered *prediction horizon*, using the *workload model*, (ii) estimate the *minimum configuration* for the prediction horizon, using the *performance model*, and (iii) possibly trigger the required *scaling action*, if current configuration differs from the minimum one.

[5] Anyway in next Chapters we evaluated this time empirically for the two considered application scenarios.

Workload Forecaster

The *Workload Forecaster* is in charge of forecasting the input rate over some prediction horizon h. It exposes a primitive *predict()* which requires as input the list w of the workloads monitored during the last *observation period*, and outputs the expected workloads $\lambda_f(h)$ during the forthcoming prediction horizon h.

The lengths of observation and prediction periods, as well as the sampling rate of observed workloads, and how many distinct workloads to forecast during the prediction horizon, have to be chosen at a preliminary stage (i.e., before starting the profiling phase). Note that the observation period w and the forecast horizon h can be quite different. Specifically, the observation period impact the accuracy of the timeseries prediction [239], while the prediction horizon is used to avoid oscillations by providing an estimation of the maximum future workload in the future h time unit. Thus, while the observation period can be very long producing a good accuracy a too long prediction horizon can lead to high over-provisioning and so it has to be tuned properly (we evaluate how to set this period in next section).

Performance Estimator

The *Performance Estimator* is in charge of estimating the performances of the target distributed service according to the *performance model* learned during the profiling phase. It exposes the primitive *estimate()* which, taking as input the expected input rate $\lambda_f(h)$ and a configuration, outputs a set of metrics describing the estimated performances of the distributed service (as expected throughput, expected CPU usage, etc.).

Decider

The *Decider* module, through the primitive *decide()*, computes the minimum configuration to sustain a forecasted workload and sends the consequent scaling actions to the Configuration Manager. The Decider leverages the Workload Forecaster to predict workloads, and the Performance Estimator to estimate the expected performances of the distributed services given a certain configuration and an expected input rate.

Configuration Manager

The *Configuration Manager* is in charge of applying a configuration. It accepts scaling actions from the Decider and applies the requested configuration to the target distributed service by activating (or deactivating) service instances.

4.3 Discussion

In this chapter we introduced PASCAL, a refinement of the MYSE architecture to be employable in a more general real-world scenario. With respect to MYSE, PASCAL employs a Service Profiler

to catch the service behaviour and substitute the Queuing model with a Performance Estimator. The main goal of PASCAL, is to provide a general architecture which act as a guideline to design an autoscaling system for a distributed service. Being PASCAL a so generic solution, in the two sequent chapters, we explain how to apply it in real systems by providing insights on two representative deployment scenarios:

- a distributed datastore where the time to complete scaling actions is potentially quite high due to the possible huge amount of data to transfer to keep the storage updated;
- a distributed stream processing system where load between resources may be temporary unbalanced due to workload oscillation.

Specifically, in the datastore scenario we will present how to cope with the transitory time after a reconfiguration due to possible huge amount of data to move. Whereas, the stream processing case study is particularly challenging as the scaling regards two dimension, i.e. resources and operators and we propose an instantiation of PASCAL to cope with it.

Chapter 5

Applying PASCAL for Autoscaling a Distributed Datastore

Typical SQL-like databases provide ACID properties (i.e. Atomicity, Consistency, Isolation, Durability), thus a sequence of operations toward such databases is perceived as a single logical operation on the data called *transaction* [104]. The Cloud Computing paradigm is enabling web-based services which need to store possibly huge amount of data. To garner such amounts of data with continuously (and possible high) input workload, *distributed datastore* are increasingly used. The CAP theorem, firstly proposed by Eric Brewer [85, 37] and then reformulated by Seth Gilbert and Nancy Lynch [96], stated as it is impossible to provide for a distributed datastore in the same time *consistency*, *availability* and *partition tolerance*. Since in a distributed system network failures are impossible to avoid, partition tolerance has to be always tolerated, thus there are two main options: ensuring consistency or availability. In the era of Big Data, important requirements for a datastore regard mostly their capability to scale and high availability. In contrast to traditional ACID guarantees, BASE (Basically Available, Soft state, Eventual consistency) semantics have been proposed [190], by introducing *eventual consistency* [227] as a weaker consistency property, but able to provide high availability.

SQL-like databases scale well vertically by increasing the hardware of hosting machine, but to scale beyond a certain point they should be distributed among a number of servers. However, such databases are not designed to function with data partitioning, and thus their distribution is a chore, whereas, NoSQL datastore can achieve an effective horizontal scalability [147] by *sharding* (or partitioning) data among available nodes instead of replicating all dataset on every node. NoSQL storage can mainly be categorised as *column-based* (e.g. Apache Cassandra [142]), *document-based* (e.g. MongoDB [169]), key-value (e.g. Amazon DynamoDB [67]), *graph-based* (e.g. Neo4J [176]) and *multi-model* (e.g. Couchbase [60]). Although most of NoSQL datastores lack ACID properties, a few databases, such as Google Spanner [59], made ACID central to their design. A main issue with horizontal scalability of datastores regards the amount of data to move which can significantly increase the time of a scaling action to keep the storage updated [142].

In this chapter we propose a solution to instantiate the PASCAL architecture for a distributed datastore. Specifically, we introduce a solution to estimate the performances and accurately tune the timing for provisioning or de-provisioning resources by taking into account the time needed to complete such operations in order to respect, with some extent, the SASO properties mentioned in Chapter 4.

Contributions. This work provides the following contributions:

- we propose an instantiation of the PASCAL architecture and provide an autoscaling solution for a distributed datastore which trigger the scaling action in order to minimise the demand point (DP) with the reconfiguration point (RP);
- we provide an experimental evaluation on a prototype integrated with Cassandra.

Chapter Structure. Section 5.1 discusses related work of scalable distributed storage; Section 5.2 defines more formally the datastore model so in Section 5.3 we can present our solution. Section 5.4 explains the implementation details of the prototype integrated with Cassandra that we experimentally evaluate in Section 5.5. Finally, Section 5.6 sums up and conclude the chapter.

5.1 Related Works

In the state of the art, autoscaling solutions are usually intended to be integrated on a web multi-tier distributed application. Except for some isolated cases where more than one tier is automatically scaled at the same time as [180], the majority of those solutions refer to one of the three tiers of a web application i.e., Web Server, Application Server, Storage Server tiers. Specific solutions for the web service layer have been already largely discussed, instead specific solutions for both SQL and NoSQL storage systems are here discussed. Some works focus on datastore scalability while providing transactional guarantees, such as Studipto et al. who proposed ElasTras [65], an elastic transactional data store which is however limited to a single partition; others sacrificed the support to distributed transactions [113]. Regarding the scaling of transactional indexes, Diegues et al. proposed STI-BT [68], a scalable transactional index based on B+Tree for Infinispan [124] with a cutoff level which split fully replicated indexes and partial replicated indexes. In [3] Al-Shishtawy et al. proposed ElastMan, an elasticity manager for key-value stores in the Cloud. Their work relies on both feedforward and feedback controllers for effective scaling actions. The main differences with our approach are that (i) they use a reactive approach and (ii) they do not consider activation/deactivation time to match the demand point with the time when the configuration is ready.

A self-managing controller for multi-tenant DBMS called Delphi has been proposed in [77]. It is based on Machine Learning techniques used to classify usage profile (e.g. disk-intensive or

CPU-intensive) of tenants sharing the DBMS, and then based on the labeling assigned to each user, it combines tenants in groups that are best suited to be placed together on the same machine since they have complementary resource utilisation. The goal is to minimise the total number of used machines, while providing acceptable performances to each tenant of the DBMS. The system learns whether a group of tenants can be placed together only based on their resource utilisation, not based on the frequency of accessing a set of partitions.

Different solutions have been proposed at different *XaaS* levels. Many works (e.g. AGILE [178]) address the scaling problem from a IaaS perspective. Likewise our solution, they profile the target system and apply AGILE to scale a Cassandra cluster. Their work is orthogonal with respect to PASCAL: indeed their goal is to minimise the start-up time of a new instance through a pre-copy live cloning approach at VM level (i.e. they act at IaaS level) while we act at the datastore level (PaaS level). Barker et al. propose ShuttleDB [27], a solution for database live migration and replication which combines VM level and database level scaling. They monitor the query latency and/or predict it to trigger scaling actions. A reactive threshold-based scaling mechanism is adopted, considering latency and resource utilisation as performance metrics. Our approach, differently from theirs, is proactive and also considers the activation time of newly provisioned nodes.

Finally, Huang et al. [119] deal with the auto-scaling and data distribution of a MongoDB datastore. Their solution scales a sharded MongoDB cluster reactively, while we operate proactively.

5.2 Distributed Datastore Model

We consider a *key-value* datastore, distributed over a cluster of nodes Each node stores a fraction of the whole data, managed by the datastore instance (i.e., service instance) running on that node. The amount of stored data is balanced among nodes. Each *value* associated to a *key* is replicated among nodes with a given replication factor α. Scaling operations lead to data migration. Following a scale-out operation, joining datastore instances receive from already active instances the portions of data they will have to managed once the scale-out will complete. Contrarily, as a result of a scale-in operation, decommissioning datastore instances transfer their data to remaining instances. We refer respectively to $T_{add}(data)$ and $T_{rem}(data)$ as the time to add and to remove a node according to the amount of $data$ stored in the database. In many real distributed datastores, adding/removing more nodes per time is not recommended because it may lead to consistency issue, especially for system based on eventual consistency, thus the best practice is to add/remove one node per time [1], hence we can consider $T_{sa} \approx T_{add|rem}(data) \cdot x$ to be the time of a scaling action that involves x service instances. Being those reconfiguration periods not negligible we consider $RP_{sa} > TP_{sa}$. We consider the distributed datastore able to sustain a workload if it maintains a throughput almost equal to the input rate, with not diverging response times. Once a cluster is no more able to sustain a workload, response times start diverging and the throughput begins to

[1] http://docs.datastax.com/en/cassandra/2.1

fall behind the input rate. Thus, with reference to our system model, we consider a distributed datastore to be in *overloaded* state if it is unable to sustain an input rate with response times under a certain $max\_threshold$, in *normal* state otherwise.

5.3 Autoscaling Solution

The main challenge in auto-scaling a distributed service lies in correctly anticipating the TP in order to make the RP matches the DP. This issue is even more important when dealing with datastore where the time required to activate/deactivate a service instance is non-negligible due to the potential huge amount of data that must be transferred to preserve consistency.

Suppose that $curr + x$ (or $curr - x$) service instances are required to handle a predicted input rate, where $curr$ is the current number of provisioned service instances. It is therefore necessary to provision (or release) x service instances by means of a proper scaling action which takes a time $T_{sa} \approx T_{add}(data) \cdot x$. In order to make $RP_{sa} \approx DP_{sa}$, our idea is to set the prediction horizon $h = T_{sa}$. During the first phase (profiling), PASCAL profiles the durations of the scaling actions according to the amount of stored data in order to pull out a *time table* which contains the time necessary to add or remove a service instance. Thus, we obtain respectively $T_{add}(data)$ and $T_{rem}(data)$ depending from the amount of $data$ stored. PASCAL besides profiles the maximum sustainable throughput $X_{max}(conf)$ for each configuration with the related response times to pull out a *performance table* containing for each configuration (i) the input rate, (ii) the throughput and (iii) the response time.

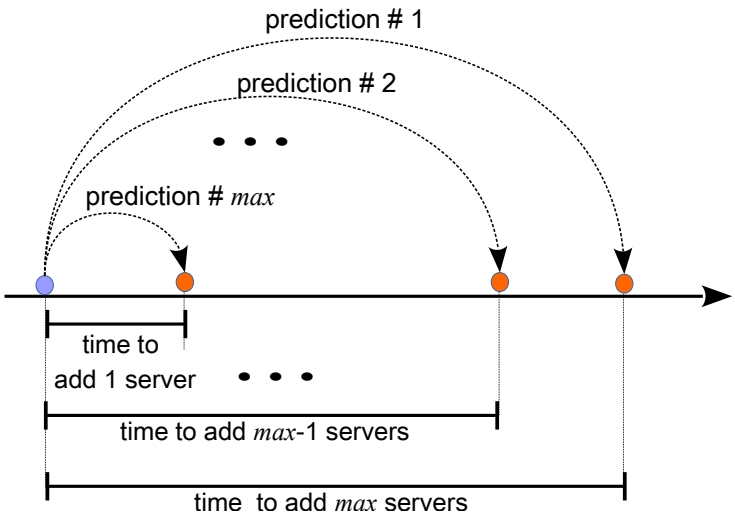


Figure 5.1: *Representation of a scaling assessment step of for both scale-out and scale-in evaluation of the datastore solution Auto-Scaling algorithm.*

In the second phase (auto-scaling) the AutoScaler invokes continuously over time the function `getMinConfig()` reported in Algorithm 1. Each configuration assessment consists of two main

parts: a scale-out evaluation followed by a scale-in evaluation. In each assessment, being known the period $T_{add}(data)$ needed to add a service instance, we are able to compute the duration of the entire scaling action T_{sa} to add a certain number of service instances. As mentioned before, we want impose $h = T_{sa}$, thus we consider t_h as the future time instant up to which forecasting the workload $\lambda_f(h)$, i.e. the maximum forecasted workload during the next horizon h. Then the Performance Estimator by relying on the performance table, is able to estimate the maximum throughput sustainable with the current configuration, so we decide whether a scale-out action is needed; otherwise, we evaluate in the same way a scale-in action.

Note that we introduced a parameter $\mathsf{p} \in (0, 1]$ referring to a max percentage of achievable maximum throughput. We call $X_B(conf) = X_{max}(conf) \cdot \mathsf{p}$ the *bounded throughput* of a configuration with $conf$ service instances. The p parameter can be set by the user according to a desired level of performance; in our solution we set p according to a maximum desiderata response time (see Sec. 5.5 for more details). Furthermore, introducing bounds on the maximum achievable throughput of each configuration allows the cluster nodes to work under their maximum processing capability, so as to handle (within a certain extent) unexpected workload spikes whose intensity is beyond what can be predicted by the Workload Forecaster.

The configuration assessment period very is low so as to execute the algorithm very frequently and maximise its efficiency in founding the right point in time to trigger scaling actions. However, to avoid oscillations, after a scaling action the Decider execution is suspended until the ongoing reconfiguration process terminates. We denote the minimum and maximum number of service instances that can be provisioned respectively as α and N [2], but note that a single scaling action is limited to $M \leq N$ for scalability reason, indeed without a limit, a cluster with hundreds of nodes should provide too far predictions. M can be set according to how far the user wants to over-provide: big values of M allow to handle workload with high variations at the cost of higher over-provisioning; on the contrary, low values of M allow more accuracy in predictions, .i.e. lower over-provisioning values at the cost of some possible under-provisioning if are not set enough new resources.

Figure 5.1 provides a graphical representation of a configuration assessment step of proposed auto-scaling algorithm; it is possible to see that the order of predictions is from the farthest configuration (i.e. the one in which we need to add/remove maximum M nodes) to the closest (i.e. the one in which we need to add/remove only 1 node) because the more servers are required to be added or removed in a single scaling action, the more time is required for the execution of such scaling action to complete. Therefore, it is required that adding or removing x servers is evaluated before the necessity of adding or removing $(x - 1)$ servers, and so on.

[2]We cannot provide less service instance than the replication factor as well we cannot provide more instance than the cluster size having one service instance per node.

Specifically in each configuration assessment we first set the current configuration (*curr*) in a variable (*conf*), then we get the last observed input workload, stored in the metric DB, through the function `getLastWorkload()` (lines 2 and 3 of Algorithm 1).

Then we start evaluating whether a scale-out action is needed (lines 4-16), therefore iteratively from the farthest (i.e. adding maximum M nodes) to the closest scaling action (i.e. 1 node), we compute the horizon h by imposing the time needed to add i service instances, thus we find out the related time instant t_h as the current time t_{now} summed to the horizon h. So, we forecast the maximum input load $\lambda_f(h)$ during the next horizon h through the `predict()` primitive exposed by the Workload Forecaster and we estimate the maximum throughput sustainable with such a configuration through the primitive `estimate()` provided by the Performance Estimator (lines 7-8). Then, we iteratively increase the number of service instances as long as we find out a configuration such that the estimation $X_{max}(conf) \cdot \mathtt{p}$ is lower than the forecasted input load $\lambda_f(h)$. If the new configuration needs M or more nodes the algorithm return $curr + M$ nodes, being M the maximum number of nodes foe a scaling action (line 11-12); if instead the new configuration requires exactly $N - curr - i$ nodes, it is returned as it is the minimum configuration to handle the expected workload in the future horizon h which require the horizon h to be effectively set (line 13-14). Otherwise no scale-out action is needed with the current horizon h.

If no scale-out action is necessary for any h is evaluated in a similar way if a scale-in action is necessary (lines 18-31). The main difference here is that we cannot neither scale-in more than M nodes nor scale under α, i.e. the replication factor, as it is the minimum number of nodes we can have in a configuration. Indeed, having less than α nodes implies that we cannot have each shard of data replicated among a sufficient number of nodes.

Finally, if neither a scale-out nor a scale-in action is necessary is returned the current configuration (line 32).

5.4 Implementation within Apache Cassandra

5.4.1 Apache Cassandra

In this work we considered Cassandra as an example of a typical distributed datastore [46]. Cassandra is a NoSQL column-oriented datastore. In Cassandra a column represents the smallest unit of storage composed by a unique name (key), a value and a time-stamp.

Data stored in the keyspace is sharded among the nodes which make up the Cassandra cluster. Each node runs a Cassandra instance that, with reference to our system model, is the service instance. In order to provide fault-tolerance and high availability, Cassandra replicates records throughout the cluster by a user-set replication factor. Cassandra offers configurable consistency levels for each single operation, providing the flexibility of trading-off latency and consistency (*ZERO*, *ONE*, *QUORUM*, *ALL* or *ANY*).

Algorithm 1 AutoScaling Algorithm for Datastore

```
1:  function GETMINCONFIG(int M, int curr, int data)
2:      conf ← curr
3:      w ← getLastWorkloads()
4:      for i ← 0 to N − curr − 1 do                          ▷ scale-out evaluation
5:          h ← T_add(data) · N − curr − i
6:          t_h ← t_now + h
7:          λ_f(h) ← predict(t_h, w)
8:          X_max(conf) ← estimate(conf)
9:          while λ_f(h) > X_max(conf) · p do
10:             conf ← conf + 1
11:             if conf == curr + M then
12:                 return conf ← curr + M
13:             else if conf == N − curr − i then
14:                 return conf
15:             else
16:                 X_max(conf) ← estimate(conf)
17:         conf ← curr
18:     for i ← 0 to curr − α − 1 do                          ▷ scale-in evaluation
19:         h ← T_rem(data) · curr − α − i
20:         t_h ← t_now + h
21:         λ_f(h) ← predict(t_h, w)
22:         X_max(conf) ← estimate(conf)
23:         while λ_f(h) < X_max(conf) · p do
24:             conf ← conf − 1
25:             if conf == curr − M then
26:                 return conf ← curr − M
27:             else if conf == curr − α − i then
28:                 return conf
29:             else
30:                 X_max(conf) ← estimate(conf)
31:         conf ← curr
32:     return curr                                           ▷ no scaling action necessary
```

Cassandra clusters can be easily scaled horizontally achieving a linear increase of performance through the *nodetool* utility [46]. During a scaling action new nodes become active after transitioning through the *joining* (scale-out) and *decommissioning* (scale-in) states. At the end of this phase a *cleanup* operation must be invoked to clean up keyspaces and partition keys no longer belonging the nodes.

5.4.2 Implementation

Service Monitor Implementation

The Service Monitor for Cassandra is made up of three sub-modules that read throughput and response times. To compute the throughput we implemented a module that leverages the `cassandra.metrics.ClientRequest` MBean. Finally, to collect the response times we implemented a module that uses the DataStax Java Driver, to interact with Cassandra keeping track of latencies.

Service Profiler Implementation

The Service Profiler is implemented with Java modules that collect through the Service Monitor the metrics of interest mentioned in the previous paragraph. The output performance model is composed by two tables mentioned in the previous subsection: the *performance table* and the *time table*. Those tables are serialised on a file that the Performance Estimator will use. From the time table the maximum values of adding and removing nodes are set as reference T_{add} and T_{rem} for the reference dataset of 1 GB used. Other $T_{add|rem}(data)$ for different $data$ are estimated through regression on the time table.

AutoScaler Implementation

The Workload Forecaster is implemented with an ANN which similar to the one proposed in Chapter 3. Before training the ANN, the trace output from the ILP needs to be filtered to smooth any rapid fluctuations; this is performed by extracting the 95^{th} percentile values related to trace subsets spanning a time interval of one hour each. In such a way we obtained an *approximated workload trace* (AWT) that better represents the service usage pattern. The AWT was then used as input training set for the ANN.

The Performance Estimator and the Decider are Java modules that implement respectively the *estimate()* and *decide()* primitives. The Performance Estimator loads the serialised files containing the two tables output from the Service Profiler to make the estimation that the Decider, by implementing the algorithm 1, will use to compute the configuration.

The Configuration Manager is a Java module that takes a scaling action *scale_out*, x or *scale_in*, x and issues the `nodetool` command to start/stop x nodes. The nodes are activated/deactivated sequentially once the redistribution during the joining/decommision phase ends.

5.5 Experimental Evaluation

5.5.1 Environment

Testbed

We evaluated the effectiveness of proposed PASCAL prototype in a computing system composed by four IBM HS22 blade servers, each equipped with two quad-core Intel Xeon X5560 2.28 GHz CPUs, 24 GB of RAM running VMware ESXi v5.1.0 type-1 hypervisor.

We deployed Cassandra nodes on dedicated VMs deployed on these blades. Load was generated through dedicated VMs and a further external Dell PowerEdge T620 server equipped with a 8-core Intel Xeon CPU E5-2640 v2 2.00GHz, 32 GB of RAM running Linux Ubuntu 12.04 Server x86_64.

Workload Trace

We evaluated PASCAL for the datastore case study case studies by using a real trace to generate the input load. Specifically, we used a subset of a 10 GB Twitter trace with 3 months of tweets captured during the European Parliament election round of 2014 from March to May in Italy. To make tests with the real trace practical, we selected a subset ranging the 41 most intensive consecutive days having high load variations, and then applied a 60:1 time-compression factor to allow the replay of the real trace with reasonable timing. For the profiling phase we instead injected for 30 minutes a stair-shaped curve.

Cassandra Cluster

We evaluated the effectiveness of the proposed prototype using a cluster composed by 6 Cassandra nodes, each one installed on a VM configured with 4 CPU cores, 4 GBs of RAM, running Linux Ubuntu 14.04 Server x86_64.

Load Generation

In order to properly assess the effectiveness of the PASCAL prototype, we developed a modified version of Apache JMeter as a workload generator able to generate massive requests against Cassandra according to a real workload trace provided as input. Each workload consists of CQL queries that request random keys. All of those reads are executed with Consistency Level `ONE`. In such a way, we are evaluating cluster performances as if it is intended to be used by a Read-Intensive application.

Config. Change	Max Time (sec)
3 to 4	155
4 to 5	155
5 to 6	155
4 to 3	120
5 to 4	145
6 to 5	160

Table 5.1: *Maximum times to add and remove a node from a configuration to another with the reference dataset of 1 GB*

Reference Dataset

We created a Cassandra keyspace with replication factor $\alpha = 3$ containing a column family with 11 columns; the first column is the primary key, while the other 10 columns contain random text strings of 20 characters for each field. The full dataset contained 1GB of data. Since we configured the keyspace with $\alpha = 3$, all experiments had a minimum number of nodes equals to 3 which have exactly the same data. By adding nodes, the 1GB of data will be sharded among the available nodes.

Parameters Setup

The Auto-Scaler prototype was configured with the following parameters indicating the minimum and maximum system configurations: $\alpha = 3$, $M = 6$, thus four possible system configurations were tested i.e. 3, 4, 5 and 6 nodes. In Figure 5.5 is represented the maximum sustainable throughput for each configurations with the related response time. These values represent the performance model output from the Service Profiler and we make use of it to properly set the bounded throughput. Specifically, the parameter p was set to 0.7, thus we allow each configuration to serve requests at a bounded throughput corresponding to the 70% of their maximum throughput. To choose this value we followed the heuristic of limiting for each configuration the response time to 4.5 ms, i.e. the *max_threshold*, that corresponds to the point where the average response time start diverging with a high standard deviation. Maximum and bounded throughput with the chosen p are reported in Table 5.5.3.

5.5.2 Reconfiguration Overhead

We evaluated the time needed to switch from a configuration to another. Collected times are reported in Table 5.5.2, thus we obtain $T_{add} = 155$ sec and $T_{rem} = 160$ sec. We then evaluated how the

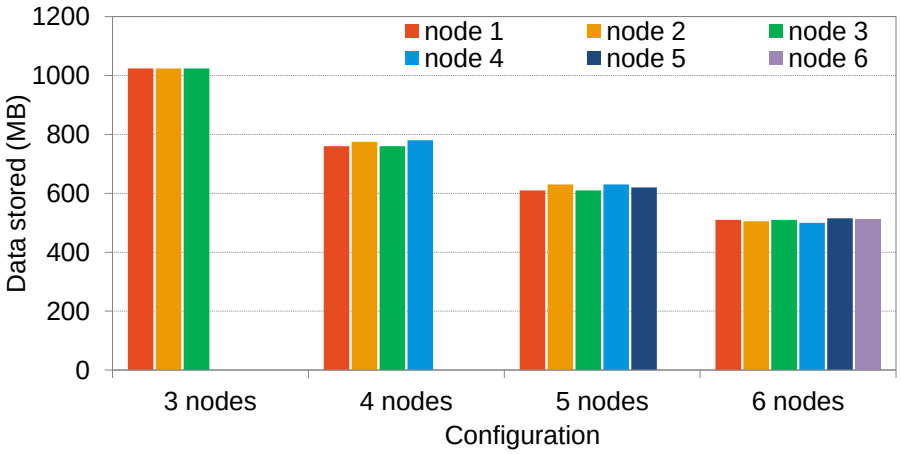


Figure 5.2: *Distribution of 1.0 GB dataset stored with replication factor 3. By increasing nodes in the configuration, data are almost equally sharded among available nodes.*

insertion/removal of a new node in the Cassandra cluster would impact the cluster performance, namely throughput, CPU utilisation and response times. In particular we studied how the system behaves when, during the *joining* and *decommisioning* process, a constant workload with input rate of 50.000 requests per seconds is submitted to the cluster.

Figure 5.3 shows the impact of the joining process. We set the cluster starting with 3 nodes and a fourth node is inserted at time 120 seconds. Before and during the joining process, each node of the cluster serves one third of the received requests. After the joining process is completed (second 210), the new node starts serving requests and as result all nodes of the cluster now serve one fourth of the received requests. The cleanup operations are triggered 60 seconds after the time instant in which the fourth node has completed its joining process and does not impact on the cluster throughput.

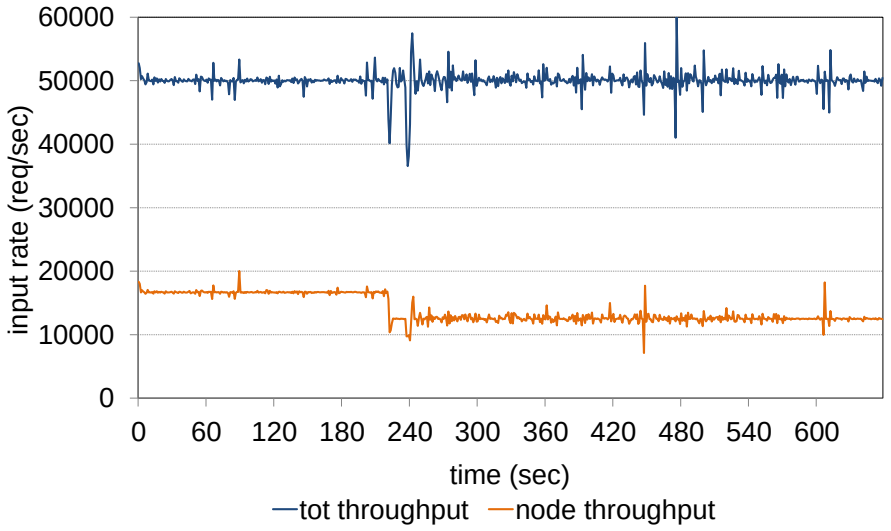


Figure 5.3: *Joining impact on throughout when input rate is 50K request/sec*

$conf$	$X_{max}(conf)$ **(tps)**	$X_B(conf)$ **(tps)**
3	92476	64733
4	105869	74108
5	118036	82625
6	131406	91984

Table 5.2: *Maximum and bounded throughputs with 70% threshold*

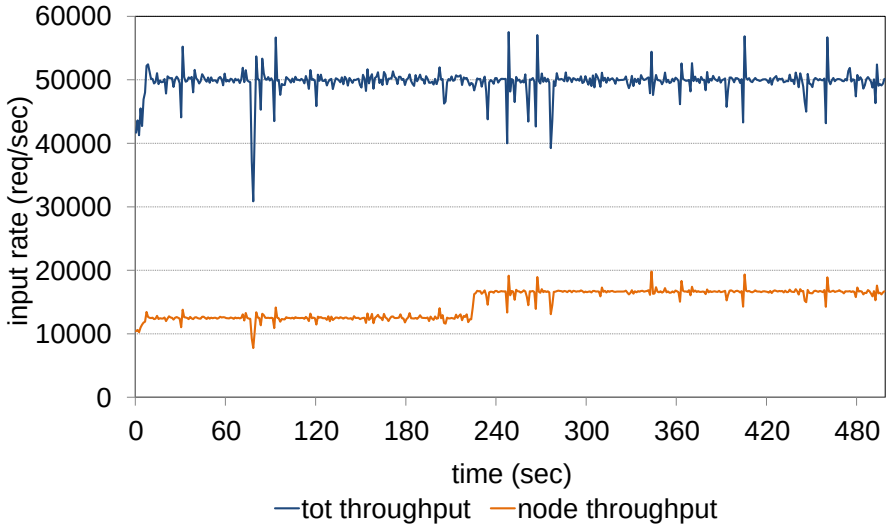


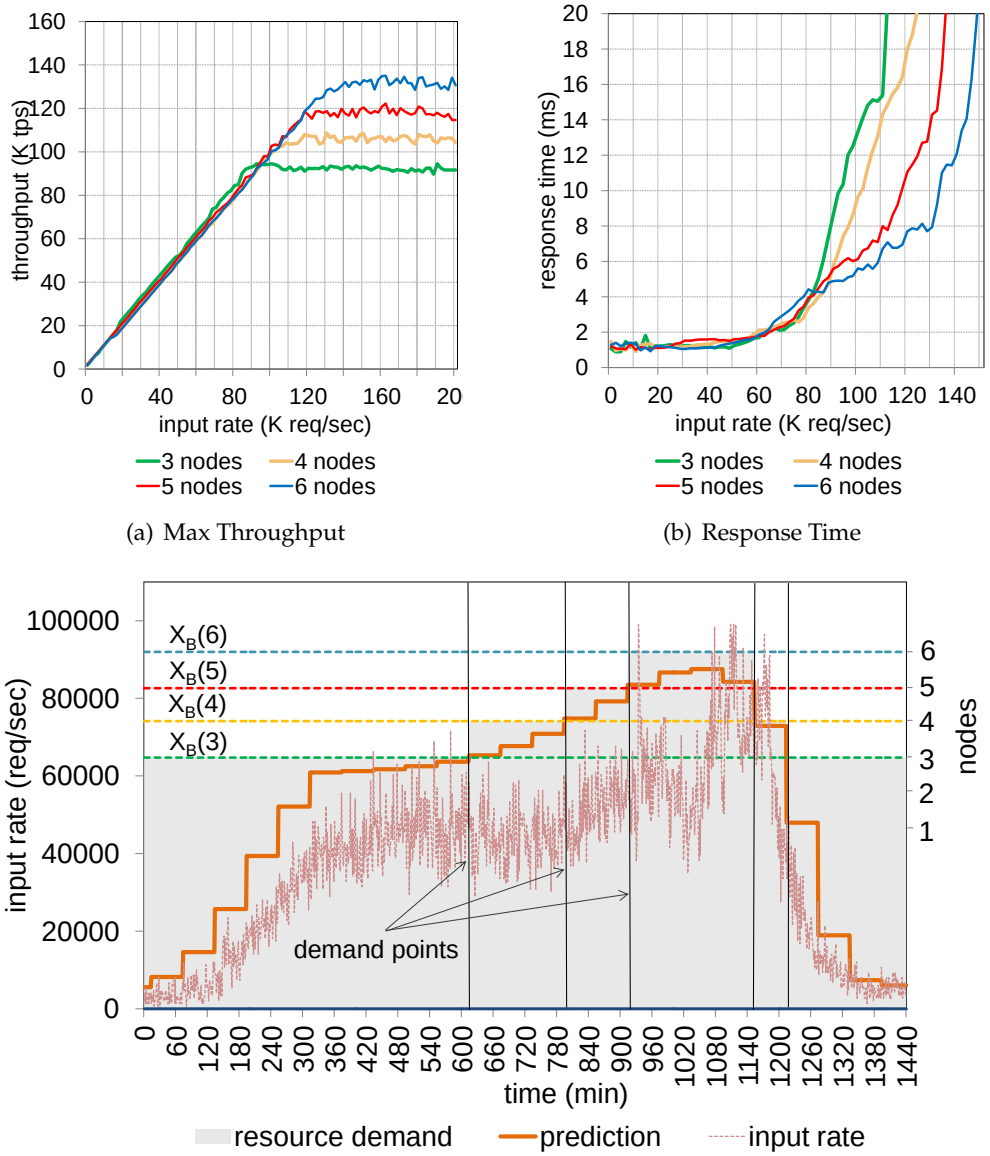
Figure 5.4: *Decommission impact on throughout when input rate is 50K request/sec*

Figure 5.4 presents the impact of the decommission process. We set a cluster starting with 4 nodes and the removal of a node is triggered at time 120 seconds. Before and during the decommission process, each node of the four, serves one fourth of received requests. As soon as the decommissioning process is completed, the removed node stops serving user requests, and its portion of handled requests is fairly redistributed among the remaining three nodes, each one serving one third of requests without any impact on the total throughput.

5.5.3 Test Results

Figure 5.5(c) reports a real input rate injected into Cassandra with the related workload prediction. Through the prediction and the bound throughput values, we obtain the resource demand to figure out which is the minimum configuration able to handle the predicted workload. The bounded throughput values are obtained from the performance model by applying the heuristic aforementioned on the maximum throughput and the response time for each configuration (see Figure 5.5(a) and 5.5(b)).

To evaluate PASCAL, we compared the performance of our system with an over-provisioning solution and a solution based on average load. In Figure 5.6 we show how the throughput sus-



(a) Max Throughput

(b) Response Time

(c) Bounded Througputs

Figure 5.5: *Performance Model obtained by profiling throughput and response times with different configurations under different input rates. Figure 5.5(a) shows the max throughput from each configuration. From Figure 5.5(b) it is possible to see how the response times diverge over the max sustainable input rate. We set a threshold to those throughput levels and we used the $X_B(conf)$ (reported in Table 5.5.3) to decide the minimum configuration. Finally, Figure 5.5(c) shows how we actually decide the configuration from the forecasted workload through the performance model.*

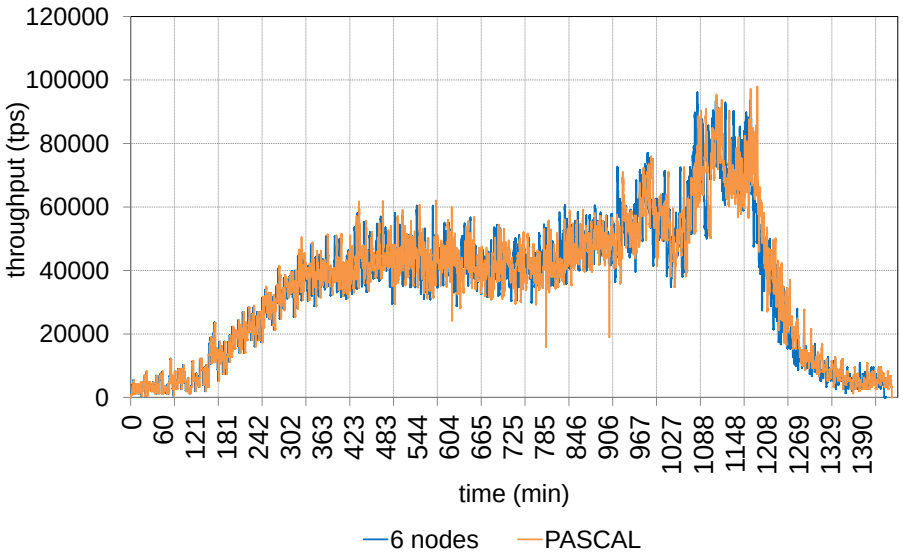


Figure 5.6: *Throughput comparison between 6-nodes config and PASCAL*

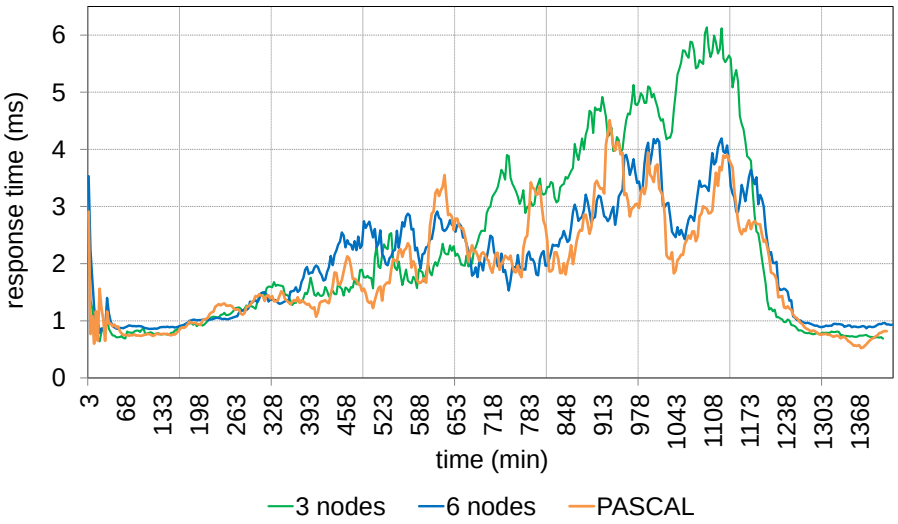


Figure 5.7: *Response time comparison between static configs and PASCAL*

action	TP	RP	DP	duration	Δ
+1	7242 s (603 min)	7347 s (612 min)	7380 s (615 min)	105 s	33 s
+1	9399 s (783 min)	9492 s (791 min)	9540 s (795 min)	93 s	48 s
+1	10848 s (903 min)	10935 s (911 min)	10980 s (915 min)	87 s	45 s
-2	13602 s (1134 min)	13833 s (1152 min)	13860 s (1155 min)	231 s	27 s
-1	14379 s (1198 min)	14514 s (1210 min)	14580 s (1215 min)	135 s	66 s

Table 5.3: *Scaling Event Times*

tained from PASCAL is the same of the over-provisioning solution (with static 6 nodes), except for a very short period of system reconfiguration appreciable mostly at minute 780 and 910. Comparing the performance in term of response times is instead possible to see how PASCAL follows the same behaviour of the over-provisioning solution, while the solution based on the average load (with 3 nodes) after the minute 600, starts increasing its response times. These results are presented in Figure 5.7. It is also possible to note between the minutes 390 and 525 as well as over the minute 1173, that PASCAL has a lower response time with respect to the over-provisioning solution. This result makes sense because PASCAL, by using less nodes with respect the over-provisioning solution, allows to have a lower overhead due to nodes inter-communication. Till minute 600 PASCAL provides a configuration with 3 nodes; having a replication factor $\alpha = 3$, when a client request a data to a random node, it always finds a node which has the requested data and it is able to send to the client directly the response. When instead a client request a data to a random node in an over-provisioning configuration having 6 nodes, with certain probability it will not find the data in that node and the latter has to forward the request toward a node which has the requested data.

From table 5.5.3 is possible to see how each scaling action is triggered (TP) so as to have the RP some times before the DP. In the same table are shown the duration of the scaling action and a value δ representing how many seconds before the DP the configuration is ready (i.e. RP). As a result we can see how each scaling action concludes 30-60 seconds before the demand point so as to avoid an excessive period of over-provisioning, and providing an adequate configuration offering good performances as aforementioned by Figure 5.7 and 5.6.

5.6 Discussion

In this chapter we proposed an instantiation of the PASCAL architecture in a distributed datastore scenario. We proposed a solution to elastically scale the datastore in order to match as more as possible the reconfiguration point with the demand point so as to minimise the over-provisioning period and avoid performance worsening. To this purpose, the proposed solution considers the time of activation/deactivation of a new service instance through a profiler which is able to estimate the duration of each scaling actions. An experimental evaluation conducted on a real prototype integrated within Apache Cassandra we showed the effectiveness of our solution. Specifically, we showed as the system reconfiguration is always ready thus to anticipate some seconds the demand point and reduce the over-provisioning. Furthermore, by employing only resources actually necessary, the instantiation of PASCAL in specific periods performs better than the over-provisioning configuration as a lower overhead due to a less message exchange between nodes allows to reduce the response time while ensuring the same throughput.

Chapter 6

From PASCAL to ELYSIUM: Elastic Symbiotic Scaling of Operator and Resources for Stream Processing Systems

Stream processing systems (SPSs) process unbounded streams of input tuples by evaluating them according to a given set of queries. Queries are usually modeled as graphs, where vertices represent processing elements called operators and edges correspond to streams of tuples moved between operators. This data processing model allows to break down complex computations into simpler units (the operators), independently parallelize them, and deploy the resulting system over any number of computing machines. Having the computation executed in parallel by several distinct operators on many machines is the core feature of *distributed stream processing systems* as IBM Spade [93], Apache Storm [216], Apache Flink [10] and many others. Such flexibility allows to scale horizontally in such a way to provide the computational power required to sustain a given tuple *input load* with a reasonable processing latency. For these characteristics, SPSs today represent a fundamental building block for a large number of big data computing infrastructures [109].

A complex challenge SPSs need to cope with is input dynamism. Such systems, in fact, are designed to ingest data from heterogeneous and possibly intense sources like sensor networks, monitoring systems, social feeds, etc. that are often characterized by large fluctuations in the input data rates. Solutions based on over-provisioning are considered cost-ineffective in a world that moves toward on-demand resource provisioning built on top of IaaS platforms.

Recently, researchers introduced the idea of *elastic* SPSs that continuously adapt at runtime to changes in the input rates, to accommodate load fluctuations by provisioning more resources only when needed. The requirements for the controller of an elastic SPS have been informally defined in [92] with the SASO properties mentioned in Chapter 4.

Several optimisations have been identified [116] and several approaches [112, 111] have been proposed to make SPSs elastically scale. These solutions scale the system by increasing operators'

parallelism (*operator scaling* or *fission*) and accordingly provisioning new computing resources (*resource scaling*). However, by looking at how SPSs work under stressing workloads, it is apparent that operator and resource scaling address two distinct aspects of a same problem. In particular, operator scaling allows to subdivide the load of the specific computation implemented by an operator, thus enabling efficient resource usage through load balancing. On the other hand, correct provisioning through resource scaling is crucial to avoid excessive contention for the execution of the operators.

In this chapter we claim and show that, although both aspects must be taken into account, they don't need to be always exercised jointly and that it is possible to build a more efficient elastic scaling solution for SPSs by accurately managing them. In particular, we advocate a "symbiotic" management of operator and resource scaling, where their independent and/or combined effects increase the global efficiency of the system.

We introduce *Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems* (ELYSIUM), a new elastic scaling solution for distributed SPSs that scales operators and resources in a symbiotic fashion to let the system work always in a correctly provisioned configuration where the least amount of resources are wasted (4^{th} SASO property). ELYSIUM can be considered as an instantiation of the PASCAL architecture for a SPS.

A tunable *assessment period* parameter allows ELYSIUM to avoid oscillations (1^{st} SASO property); ELYSIUM first adapts the parallelism for each operator used by the application to avoid bottlenecks on operator instances. Then it checks if the current resource provisioning is the smallest that will let the system work without incurring any performance degradations. For this last check, ELYSIUM leverages the PASCAL's Performance Estimator with a novel approach to compute the expected resource consumption, given an input load and a configuration, so as to accurately and quickly adapt to the workload in a single reconfiguration (2^{nd} and 3^{rd} SASO properties). The PASCAL's Service Monitor system lets ELYSIUM collect at runtime fine-grained information on resource usage that is then used to decide how the system must be scaled. With this approach ELYSIUM can scale independently operators and resources as well as jointly scale them, whenever this is needed.

Contributions. The content of this chapter has been published in [153]. The novelty of ELYSIUM can be summarised as follow:

- we explain why operator and resource scaling impact on two distinct aspects of SPSs scalability, and propose how to symbiotically manage them to elastically provision the system in a more efficient way;
- we propose a reactive/proactive elastic scaling solution for SPSs that consider operator and resource scaling as two distinct solutions that need to be combined only when necessary; ELYSIUM employs a fine-grained model of resource usage to estimate how the SPS will

behave under a given load, which enables to properly choose how many instances (for each operator) and resources to set;

- we provide an in-depth evaluation of ELYSIUM's performance by testing a prototype on real stream processing applications under different workloads and comparing it with a standard elastic scaling solution employing only the joint scaling approach.

Chapter Structure. Section 6.1 presents related works; 6.2 defines more formally the system model and the problem to tackle, so that in Section 6.3 we can present our approach. Section 6.4 presents the ELYSIUM implementation on Apache Storm, while the experimental evaluation is described in Section 6.5. Finally, Section 6.6 sums up the work and points out future work.

6.1 Related Works

How to scale SPSs has been extensively studied and analysed from an application-level perspective by Hirzel et al. in [116]. Most of the works that tackle this problem at the application level [238, 102, 92] assume that a fixed amount of computing resources are available, and then strive to define the best allocation of operators to such resources. From this point of view, ELYSIUM works on a fully orthogonal direction, as we assume that a possibly infinite amount of resources is available (like in a cloud computing scenario), but aims at consuming the minimum amount needed to run the SPS with the goal of being cost efficient. Differently from previous solutions, ELYSIUM manages operator and resource scaling in a symbiotic fashion, deciding which one to apply or whether to apply both depending on the specific scenario.

A large fraction of solutions available in the literature scale one resource at a time. A notable exception is represented by [92], where the authors propose *rapid scaling* i.e. a solution that reduces the number of iterations needed to reach the target configuration. ELYSIUM further improves along this same direction by providing a solution that removes/provisions multiple resources in a single scale-in/out action on the basis of the resource usage estimated from either current or predicted input load, depending on whether the reactive or proactive mode is enabled.

Heinze et al. in [112] presented a solution to perform horizontal scaling according to the workload pattern evolution and by optimising a cost function. Such prototype extends the FUGU stream processing system [111], where the authors compared threshold-based techniques with reinforcement learning techniques as defined in [154]. Furthermore, in [110] they proposed a latency aware solution. A notable work for complex event processing has been proposed by Koldehofe et al. []; they presented a solution to dynamically increase the parallelisation degree in an autonomic manner to limit the buffering imposed on event detection in the presence of dynamic changes to the workload.

ELYSIUM, with respect to the previous solutions, is able to predict large load fluctuations and thus allows scale-in/out of multiple instances and resources at the same time.

While the majority of the works are reactive and based on thresholds, i.e. they act after an overload/underload detection, Ishii et al. in [127] proposed a proactive solution to move part of the computation to the cloud when the local cluster becomes unable to handle the predicted workload. The proactive model we propose is more fine-grained thanks to a resource estimator that allows to accurately compute the expected resource consumption given an input load and a configuration.

Recently, some efforts have also been spent to consider together problems related to elasticity and fault tolerance. In [48], the authors considered the problem of scaling stateful operators deployed over a large cloud infrastructure. In cloud environment, in fact, failures are common and managing replicated operators in presence of crash and recoveries introduces additional overheads with respect to those imposed by automatic scaling. The scaling strategy they propose, contrarily from us they (i) used a joint approach hence the detection of an overloaded operator leads to the allocation of new resources, and (ii) scale one unit per time.

6.2 Distributed Stream Processing Model

We model a computation in a SPS as a directed acyclic graph where vertices represent *operators* and edges represent *streams* of tuples between pairs of operators. We define such a graph as an *application* and there can be more applications running in a SPS. Each operator carries out a piece of the overall computation on incoming tuples and emits downstream the results of its partial elaboration. In general, an operator has n_i input streams (0 for source operators) and n_o output streams (0 for sink operators). An application is also characterized by an input load that varies over time and represents the rate of tuples fed to the SPS for such application (*input rate*). Each input tuple generates multiple tuples that traverse several streams in the application graph. The processing of some of these tuples may possibly fail; in this case we say that the tuple is *failed*. Conversely, if all the tuples generated in the graph are correctly processed, then we say that the corresponding tuple is *acked*. The rate of tuples that are acked over time is referred to as *throughput*. Each acked tuple takes a certain amount of time to complete, which we refer to as *latency* (i.e. response time in the PASCAL's system model described in 4.1), measured from the time at which the tuple was fed into the SPS until the moment it gets acked.

For the sake of simplicity, and without loss of generality, we assume that a stream connecting operators A (upstream operator) and B (downstream operator) can be uniquely identified by the pair (A, B), which means that no two distinct streams can connect the same pair of operators. The *selectivity* for a stream (A, B) is defined as the ratio between the tuple rate of (A, B) and the sum of the tuple rates of all the input streams of A, i.e. the selectivity of (A, B) measures its tuple rate as a function of the total input rate of A [116]. In this paper we assume to work with SPS applications having constant average operators' selectivities at runtime, similarly to applications presented in [129, 23, 207].

To let the application scale at runtime each operator can be instantiated multiple times such

that its instances will share the load provided by the input stream. However, the maximum number of instances for an operator op is upper-bounded by an application parameter max_parall_{op}. Each operator instance runs sequentially and uses a single CPU core at a time[1]. The number of available instances for a given operator is defined as its level of *parallelism*. As a consequence, a stream (A, B) can be constituted by several sub-streams, each connecting one of the instances of operator A to one of the instances of operator B. To simplify the discussion, we assume that the SPS is able to fairly distribute the load among the available instances of each operator. This is achieved by means of grouping functions that manage how tuples in a stream are mapped in its sub-streams [197, 198, 175].

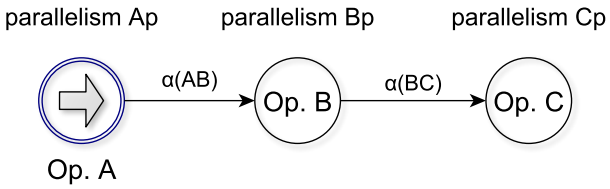


Figure 6.1: *SPS computation model*

When an application is run, the SPS uses a *scheduler* to assign the execution of each single operator instance to a *worker node* among the many available in a computing cluster.

As stated in the PASCAL system model in Section 4.1, we recall that we are assuming worker nodes in the cluster to be homogeneous (same configuration for CPU cores, speed, memory, etc.) and can be activated on-demand as for an IaaS provider.

The *configuration*, includes the number of used worker nodes, as in the PASCAL system model, and the parallelism for each operator for each running application.

We define three possible states for a worker node at runtime by comparing its CPU usage to two thresholds ($cpu\_min\_thr < cpu\_max\_thr$):

- *cpu_low*: CPU usage $<$ *cpu_min_thr*
- *cpu_avg*: *cpu_min_thr* $\leq$ CPU usage $\leq$ *cpu_max_thr*
- *cpu_stress*: CPU usage $>$ *cpu_max_thr*

Similarly, we define three possible states for an operator instance at runtime by comparing its CPU core usage to two thresholds ($core\_min\_thr < core\_max\_thr$):

- *oper_low*: core usage $<$ *core_min_thr*
- *oper_avg*: *core_min_thr* $\leq$ core usage $\leq$ *core_max_thr*
- *oper_stress*: core usage $>$ *core_max_thr*

[1]This operator execution model is common to several SPSs like Apache Storm [216] and Apache Flink [10].

The configuration of a SPS is *correct* (which refers to the *normal* state of PASCAL system model) as long as no operator instance and no worker node is in the *stress* state (which refers to the *overloaded* state of PASCAL system model). Note that we are here considering CPU-bound applications. A more complete model that considers memory and bandwidth consumption is subject of our future work. We recall from PASCAL system model that, since we are assuming a homogeneous cluster, we consider that the *minimum configuration* is the correct one having the minimum number of worker nodes required to sustain a certain input load.

The SPS can be scaled by tweaking the operator parallelism or the number of available worker nodes. The PASCAL system model is so extended giving the possibility to scale along two dimensions. These two operations can be performed independently as they address two different issues: operator overloading and scarceness of computing resources, respectively. In some cases, increasing the level of parallelism for an operator, i.e. increasing the number of instances for that operator, may also impact resource provisioning demanding for further working nodes. Furthermore, we cannot exclude that in some (unfrequent) cases scaling up the parallelism of an operator may possibly induce a reduction of resource provisioning.

We consider that reconfigurations have a cost (*reconfiguration overhead*) due to (i) the elastic controller execution and (ii) a period of performance degradation whose amplitude and duration are mainly related to:

- R_{state}, i.e. the operator state migration time;
- $R_{restart}$, i.e. the time due to topologies restarting;
- R_{queue}, i.e. the time to process tuples queued during $R_{state} + R_{time}$.

These time periods strictly depend on the specific strategies employed by the SPS to handle application reconfigurations at runtime.

The problem we tackle in this work is how to choose, at runtime, configurations for a SPS in such a way that all will be correct despite variations in the input load (i.e. number of tuples per second injected in the system). Ideally, these configurations should also be minimal but we cannot guarantee such a property.

6.3 ELYSIUM Autoscaling Solution

6.3.1 Symbiotic Scaling Strategy

ELYSIUM is based on the following idea: stress at the operator instance level and stress at the worker node level are two different issues that can be addressed by separately scaling-in/out operators and worker nodes. In some cases, the two issues are interrelated in such a way that both operators and worker nodes will be scaled-in/out.

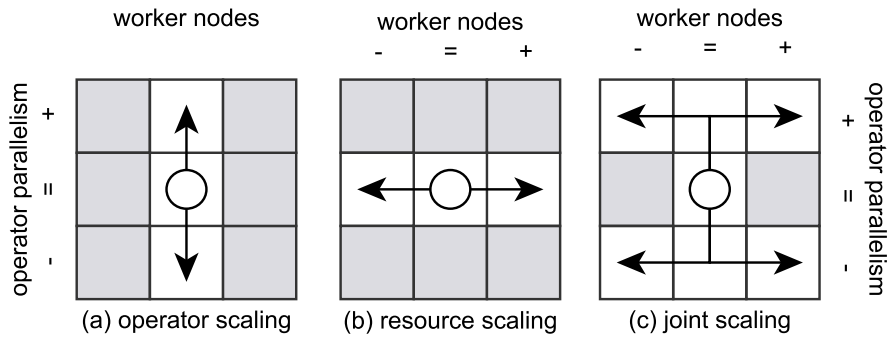


Figure 6.2: *Scaling Options in a Distributed Stream Processing System*

Figure 6.2 depicts the different scaling strategies used by ELYSIUM. Figure 6.2(a) shows the operator scaling operation where for one or more operators the number of parallel instances is decreased or increased. This strategy can be adopted when an operator instance is in a *oper_stress* status, as this may indicate that a single instance is saturating a CPU core because it is overloaded by incoming tuples. By increasing the operator parallelism we increase the probability that its load will be shared among other instances, thus alleviating its stress state.

Figure 6.2(b) shows the dual *resource scaling* operation performed to scale-in/out resources by adding or removing worker nodes assignable by the SPS scheduler. This strategy can be adopted when one or more worker nodes have their CPU in a *cpu_stress* status, as this may indicate that the resources available to the SPS scheduler are insufficient to handle the global application input load. By increasing the amount of available resources we decrease stress on pre-existing worker nodes, thus allowing the SPS to ingest more data for the application.

Finally, Figure 6.2(c) shows a *joint scaling* operation where resources and operators are scaled-in/out together. This strategy can be adopted when the scale-out of one or more operators saturates available resources, thus requiring a resource scale-out operation. This is the strategy employed by most of the elastic scaling solutions for SPS present in the state of the art (see Section 6.1). The picture shows that we don't rule out the possibility of scaling-in resources after having scaled-out processes (and vice-versa). These counter-intuitive scenarios may arise in specific setting where, for example, after an operator scale-out decision, the SPS scheduler is a able to better distribute instances over the available worker nodes, thus reducing the global load on the cluster.

6.3.2 ELYSIUM Architecture

ELYSIUM profiles the SPS and the applications running on top of it with the aim of producing accurate estimations about the resource consumption a specific configuration can cause given a certain input load. By leveraging such estimations, ELYSIUM periodically calculates a new configuration to be adopted by the SPS during the next *assessment period*. This calculation is performed

striving to minimise the number of used worker nodes, while providing a configuration that will be correct with high probability for the whole duration of the next period. The assessment period can be tuned depending on the specific cluster characteristics, and accordingly to the desired tradeoff between (i) the need to reduce the amount of time the system will run in a non correct configuration, and (ii) the reconfiguration overhead caused by adopting each new configuration.

ELYSIUM can be used either in *reactive* or *proactive* mode. The difference lies in the input load used for the estimations: if the real current input load is used, then ELYSIUM scales *reactively*, otherwise, if input load is forecasted over a certain prediction horizon, then ELYSIUM scales *proactively*. In the former case the assessment is performed such that the new configuration is correct with respect to the recently observed input load. Conversely, in the latter case ELYSIUM uses the maximum predicted input load for the next assessment period as a metric to identify correct configurations.

While working in reactive mode, ELYSIUM profiles the applications to learn what would be the CPU usage for the worker nodes in a certain configuration when a given input load is fed to the SPS. This is accomplished by splitting ELYSIUM execution in two phases: a *profiling phase*, where it learns these information, and an *autoscaling phase*, where it makes periodical assessments leveraging learned application profile. While working in proactive mode, the profiling phase also includes an input load learning step used to enable load prediction.

ELYSIUM's architecture (Figure 6.3) includes three subsystems: (i) a Monitoring subsystem which collects and provides the metrics required to carry out the two phases, (ii) an Application Profiler subsystem implementing the phase 1 and (iii) an AutoScaling subsystem for the phase 2.

Monitoring Subsystem

The Monitoring subsystem consists of a set of *monitoring agents* deployed over the worker nodes and a *metric DB* where metrics are stored. Each metric agent monitors the operator instances running on the same worker node where it is deployed, collects metrics and periodically stores average values computed over a sliding time window into the metric DB. Collected metrics are (i) inter-operator instance traffic, measured as the tuple rate for each pair of communicating operator instances, (ii) CPU usage of each operator instance and (iii) CPU usage of the whole worker node due the SPS. In proactive mode, also the input load is collected, and it is measured as the tuple rate in input to each application running in the SPS.

Application Profiler Subsystem

While working in reactive mode, it includes three distinct profilers, each aimed at learning a specific aspect of an application: (i) the *Selectivity Profiler* (SP) learns the selectivity of each operator

(see Section 6.2), (ii) the *Operator CPU Usage Profiler* (OCUP) learns how the CPU usage of each operator instance varies as a function of its input rate and (iii) the *Overhead Profiler* (OP) learns how the CPU usage of a worker node varies depending on the sum of the CPU usages of its operator instances. The latter is required as, typically, SPSs impose some overhead over running applications to provide basic services like process management, message queue control threads, etc. Therefore, the worker node total CPU usage is the sum of the usage imposed by running operator instances and the overhead. While working in proactive mode, a further *Input Load Profiler* (ILP) is used, to learn input load patterns over time.

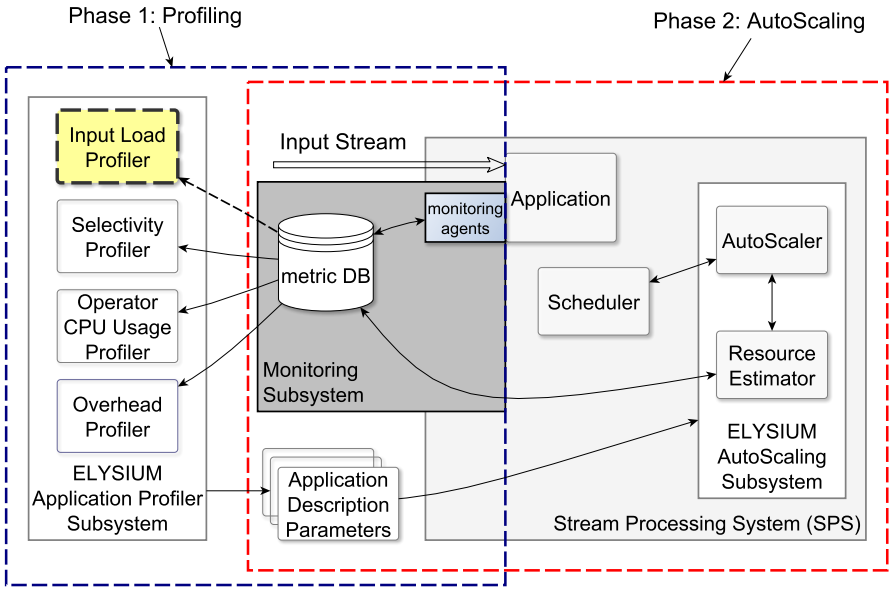


Figure 6.3: *ELYSIUM Architecture integrated in the SPS. The dotted blue line indicates modules involved in the first phase of application profiling, the red dotted one those involved in the second phase of autoscaling. The Input Load Profiler, represented with a yellow background, is used only when switching from reactive to proactive mode.*

The outputs from the profilers constitute the *application description parameters* (see Figure 6.3) that will be used by the AutoScaling subsystem to estimate the state of worker nodes and operator instances (see Section 6.2). In the following, we detail the profilers and data they collect. For the sake of simplicity, the formalisms used to model managed data don't include applications' and operators' identifiers when they are obvious.

- **SP** — Extracts metrics related to inter-operator instance traffic from the metric DB to create a dataset with records in the form $\langle up\_op, dn\_op, tuple\_rate \rangle$, where *tuple_rate* is the average tuple rate of the stream from *up_op* upstream operator instance to *dn_op* downstream operator instance. The output of the SP is the selectivity for each stream, as defined in Section 6.2.
- **OCUP** — Retrieves data from the metric DB to create a dataset having records for each operator instance structured as $\langle tuple\_rate, cpu\_usage \rangle$, where *tuple_rate* is the average input rate of the

operator instance, and *cpu_usage* is the CPU usage (in Hz[2]) that the worker node needs to run the operator instance. The output of the OCUP is a function for each operator that, given the input rate, returns the expected CPU usage that one of its instance entails.

• **OP** — Reads the metric DB to extract a dataset consisting of records in the form $\langle$*cpu_usage_ops*, *cpu_usage_sps*$\rangle$. Each record maps the sum of CPU usages of all operator instances running on a worker node (*cpu_usage_ops*) with the CPU usage of that worker node (*cpu_usage_sps*). Its output is a function that returns the expected CPU usage of a worker node due to the SPS overhead, given the sum of CPU usages due to the operator instances running on it.

• **ILP** — Profiles input load over time for each running application, on the base of data extracted from the metric DB. Such dataset includes records in the form $\langle ts, input\_load \rangle$, where *input_load* is the average input load observed during one minute[3] starting at timestamp *ts*. The output of the ILP is a function that returns the maximum input load expected during the next prediction horizon, given in input the day of the week, the hour, the minute, and the input loads seen in the last n_{ILP} minutes, where n_{ILP} is a configuration parameter whose value must be tuned empirically. This kind of input enables a combination of prediction approaches: one simply based on current time (the day of the week, the hour, the minute) to profile periodic trends, and another based on time series (input loads seen in the last n minutes) to catch behaviours depending on patterns.

AutoScaling Subsystem

The AutoScaling subsystem starts to work once the profiling phase ends, so that it can leverage the application description parameters provided by the Application Profiler subsystem. It includes two components: (i) the *Estimator*, which uses fresh data from the metric DB to compute functions provided by the profilers so as to expose methods for obtaining estimations and predictions on specific applications, and (ii) the *AutoScaler*, which starts the assessments and leverages these Estimator's methods to decide the new configuration to use.

The Estimator exposes four methods:

• `getOperatorInputRate()` — Traverses the application graph and uses operator selectivities obtained by the SP to compute the expected operator input rates starting from the application input load.

• `getOperatorInstanceCpuUsage()` — Estimates CPU usage of operator instances by dividing the total expected input rate of an operator by its parallelism and then using this value to feed the profile function returned by OCUP.

• `getCpuUsages()` — Provides an estimation of the CPU usage of worker nodes given (i) the allocation of operator instances to worker nodes provided by the SPS scheduler and (ii) expected CPU usage for all operator instances. The estimation for a given node is obtained by summing the

[2]Using Hz as a metric for CPU usage allows our system also to support heterogeneous nodes.

[3]We considered a one minute granularity for collecting input load data as this value, from our experimental evidence, provided the best compromise for input load predictions [239].

CPU usage of operator instances running on it, and then feeding this value to the profile function returned by OP.

• `predictInputLoad()` — This methods is used only in proactive mode and returns the *maximum* input load predicted for an application for the next prediction horizon. It is implemented by computing the function provided by the ILP on the inputs obtained from the metric DB.

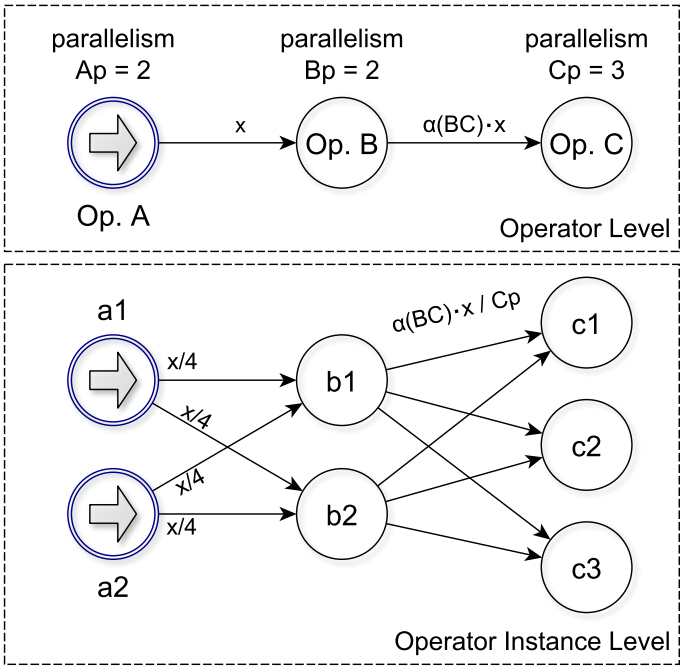


Figure 6.4: *Example of Estimator's functioning on a 3 operator application. The Estimator, through the method* `getOperatorInputRate()`*, starting from an input load x, traverses the application graph by using the selectivities provided by the SP to compute the input rate of each operator; in the figure above the input rate of the operator B is x, while the input rate of the operator C is αBC · x, where αBC is the selectivity of the stream BC. Through the method* `getOperatorInstanceCpuUsage()` *the Estimator first obtains the input rate of each operator instance by dividing the input rate of each operator by its parallelism (figure below), then, by using the function provided by the OCUP, it infers the operator instance CPU usage. Finally, through the method* `getCpuUsages()` *the Estimator infers the CPU usage of each worker node by taking from the SPS scheduler an allocation of operator instances on worker nodes. In proactive mode the input load x is predicted though the* `predictInputLoad()` *method.*

Matching with the PASCAL Architecture

It is clear to note as the ELYSIUM architecture is an instantiation of the PASCAL architecture. Specifically, the Monitoring Subsystem corresponds to the PASCAL's Service Monitor. The Application Profiler composed by the four submodules corresponds to the PASCAL's Service Profiler and the output functions of the profilers compose the Application Description Parameter which match with the workload and performance model of PASCAL. The ELYSIUM's Resource Esti-

mator instantiate the Workload Forecaster and the Performance Estimator of PASCAL and the ELYSIUM AutoScaler instantiate the PASCAL's Decider.

6.3.3 Symbiotic Scaling Solution

Figure 6.4 shows an example of how the Estimator works. The AutoScaler module works by invoking periodically its `computeConfig()` method (reported in Algorithm 2) accordingly to the configured assessment period. This operation allows to choose the configuration to apply in order to efficiently sustain an expected input load during the next assessment period. It takes as input (i) a reference to the Estimator component, (ii) a reference to the SPS scheduler used to compute allocations of operator instances to worker nodes, (iii) the list of applications currently running in the SPS, and (iv) the corresponding input loads. In reactive mode, these input loads are directly read from the metric DB, while in proactive mode they are predicted by the Estimator and obtained by calling the `predictInputLoad()` method for each running application.

The computation of a new configuration is performed by two consecutive stages. Firstly, the parallelism of each operator is adapted to avoid any CPU core overloading or under-utilisation (*operator parallelism scaling*). Then, the minimum number of worker nodes is identified to run all the operator instances without saturating the CPU of any worker node (*resource scaling*). Each stage decides a scaling action along a different dimension, and the second one takes into account the possibly updated operators' parallelism decided in the first stage.

The first stage (lines 2-9 of Algorithm 2) analyses for each running application a_k the status of their operator o_i.

Estimator's methods `getOperatorInputRate()` and `getOperatorInstanceCpuUsage()` are invoked to evaluate the CPU load that each of the p_i instances (where p_i is initialized to 1) of o_i would produce on the CPU core where it is running, given the input load for a_k. Since it is assumed that the input rate of an operator gets equally split among its instances (see Section 6.2), the value of p_i can be adjusted by increasing it in case of core overloading (estimated CPU core load greater than *core_max_thr*), or decreasing it in case of core under-utilisation (estimated CPU core load lower than *core_min_thr*), until a steady point is reached, i.e. operators in state oper_avg.

In the second stage (lines 10-16 of Algorithm 2), multiple potential configurations, each differing for the number of used worker nodes are checked. The process starts by checking the configuration with the least number of workers nodes (i.e. 1 node) and proceeds by increasing the worker nodes one at a time until a configuration is found that has no bottleneck: this is the configuration that will be used in the next assessment period. For each configuration, the scheduler is requested to produce an allocation, which is a mapping of the operator instances of running applications to the worker nodes of the configuration to test. Such an allocation and the input load of each application are passed to the `getCpuUsages()` method exposed by the Estimator, and the list of

CPU usages of the worker nodes in the configuration being checked is obtained. If any of such worker nodes is in stress state, then the current configuration does not contain enough available resources for the computation; one more worker node must be added and the new configuration needs to be checked again. Conversely, the number of worker nodes with the number of instances for each operator of the submitted applications is returned.

Algorithm 2 Symbiotic AutoScaling Algorithm

```
1: function COMPUTECONFIG(Estimator E, Scheduler S, List⟨Application⟩ apps, List⟨int⟩ input_loads)
2:     for all application a_k in apps do
3:         for all operator o_i in a_k do
4:             ir_i ← E.getOperatorInputRate(input_loads_k, o_i)
5:             p_i ← 1
6:             while E.getOperatorInstanceCpuUsage(o_i, ir_i/p_i) > core_max_thr & p_i < max_parall_{o_i} do
7:                 p_i ← p_i + 1
8:             while E.getOperatorInstanceCpuUsage(o_i, ir_i/p_i) < core_min_thr & p_i > 1 do
9:                 p_i ← p_i − 1
10:    worker_nodes ← 1
11:    while true do
12:        allocation ← S.allocate(apps, worker_nodes)
13:        cpu_usages ← E.getCpuUsages(allocation, input_loads)
14:        if ∀x ∈ cpu_usages : x ≤ cpu_max_thr then
15:            return worker_nodes, {p_i}
16:        worker_nodes ← worker_nodes + 1
```

6.4 Implementation within Apache Storm

6.4.1 Apache Storm

In this section we describe how we implemented each component of ELYSIUM and how we integrated it into Apache Storm [216], a widely adopted framework for distributed SPS. The way ELYSIUM is integrated with Storm is shown in Figure 6.5 and described in following subsections.

In Storm jargon applications, called *topologies*, are represented as acyclic graphs of operators, called *components*. Source components are called *spouts*, while all the others are named *bolts*. Spouts usually wrap external data sources and generate the input load for applications. At runtime, each component is executed by a configurable number of threads, called *executors*, which are the instances of the operators.

Storm does not provide support for stateful operator migration at runtime. For this reason, in

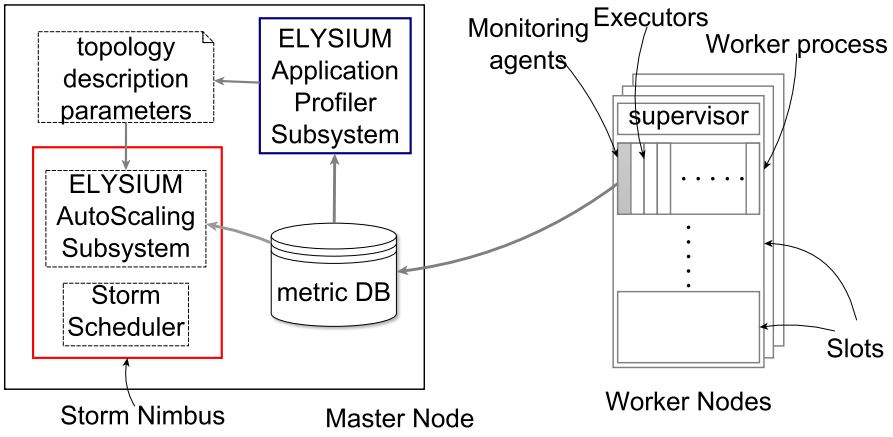


Figure 6.5: *ELYSIUM deployment in Storm*

this implementation we consider $R_{state} = 0$.

A Storm cluster comprises a single master node (*Nimbus*) which coordinates all the other nodes each locally managed by a special process called *Supervisor*. Each Supervisor provides a fixed number of Java processes (*workers*) to run executors.

A topology can be configured to run over a precise number of workers. The Nimbus is in charge of deciding the allocation of executors to available workers by running a *scheduling* algorithm. Application developers can use the embedded *even scheduler*, provided by Storm, or implement custom allocation strategies through a generic *scheduler interface*.

As a rule of thumb, each topology should use a single worker per supervisor in order to avoid the overhead of inter-process communication. Indeed, the default scheduler strives to choose the workers for a topology in such a way.

The Nimbus also provides a *rebalance* API to dynamically vary (i) the number of workers a topology can use to run its executors (*resource scaling*), and (ii) the number of executors for each component (*operator scaling*). This operation takes a time (modeled as joining/decommissioning time in PASCAL at Section 4.1) that we can consider as $RP = TP + R_{restart}$, while a further R_{queue} will be necessary to bring again the SPS in a steady state.

6.4.2 Implementation

Monitoring Subsystem Implementation

The monitoring agents are threads that run inside the workers and monitor executor metrics by leveraging Storm's metrics framework. With reference to Section 6.3.2, monitored metrics are (i) the rate of tuples received by bolts (to monitor inter-operator traffic), (ii) the CPU usage of the executors, (iii) the CPU usage of the workers, and (iv) the rate of tuples emitted by spouts (to monitor the input load). Our prototype stores every 10 seconds into an Apache Derby DB [66] (the metric DB hosted on the Nimbus) average metric values computed over a sliding window of

1 minute.

Application Profiler Subsystem Implementation

Profilers are implemented as standalone Java applications. They access every 10 seconds the metric DB to extract the required data and build a dataset. Once the profiling phase ends, they produce the output functions and store them as Java objects serialised to file.

The SP provides the list of selectivities for each stream in the topology by averaging over time collected selectivities. This approach is motivated by the initial assumption on constant selectivities. Tests reported in Section 6.5 show that SP provides reliable predictions for selectivities also with real workloads that show little selectivity oscillations.

As output OCUP, OP and ILP produces *Artificial Neural Networks* (ANNs) through Encog [108]. Specifically, the OCUP employs an ANN for each component of the topology, each one having a single input node for the input rate, and a single output node with the estimated CPU core usage. Similarly, the OP employs an input node for the sum of CPU usages due to executors and an output node with the estimated CPU usage of the worker node. The ILP employs a different ANN that takes as input the day of the week, the hour, the minute and the input loads seen in the last n_{ILP} minutes. More details about ANNs' setting and their training are discussed in Section 6.5.

AutoScaling Subsystem Implementation

The AutoScaling subsystem is implemented as a Java library to be imported by the Nimbus. It implements the scheduler interface, and the Nimbus is configured to be invoked periodically with period equals to the chosen assessment period. In this way, assessments are executed at the right frequency and have access to all the required information about allocations.

The Estimator is a Java object that accesses the metric DB and implements the methods introduced in Section 6.3.2. It loads the profiles produced by the Application Profiler subsystem by unserialising them.

The AutoScaler is the Java object that implements the scheduler interface and executes the autoscaling algorithm. It wraps the default scheduler of Storm and uses it to simulate allocations when checking the effectiveness of configurations. In case the chosen configuration is different from the current one, it issues a rebalance operation through the Nimbus API to apply the new configuration, that is to assign (i) a different number of workers to a topology, and (ii) a different number of executors to each component.

6.5 Experimental Evaluation

6.5.1 Environment and Deployment

Testbed

The environment used to deploy and test ELYSIUM was composed by 4 blade servers *IBM HS22*, each equipped with 2 Quad-Core *Intel Xeon X5560* 2.28 GHz CPUs and 24 GB of RAM. We distributed the Storm framework on a cluster of 5 VMs, each equipped with 4 CPU cores and 4 GB of RAM. One was dedicated to hosting the Nimbus process and the *Apache Derby DB*, while the remaining 4 hosted the worker nodes. One further VM was used for the Data Driver process, in charge of generating the input load. This VM was equipped with 2 CPU cores and 4 GB of RAM. The Data Driver process generates tuples according to a given dataset, then sends them to a *HornetQ* [118] Java Messaging Service (JMS) queue. The spouts are connected to such JMS queue to get the tuples to inject into the topology.

Reference Applications and Dataset

To evaluate ELYSIUM we implemented two topologies that we refer to as T1 and T2 respectively: T1 performs a *Rolling-Top-K-Words* computation [159] and T2 implements *Sentiment Analysis* [123]. Each operator in the topologies has a parallelism in the range $[1;4]$.

We evaluated ELYSIUM by using both synthetic and real traces to generate the input load. As synthetic traces, we employed (i) a stair-shaped curve, (ii) a sine function, and (iii) a square wave. As real trace we used a subset of a 10 GB Twitter dataset containing 3 months of tweets captured during the European Parliament election round of 2014 from March to May in Italy. To make tests with the real trace practical, we applied to them a $60 : 1$ time-compression factor to allow the replay of the real trace with reasonable timing.

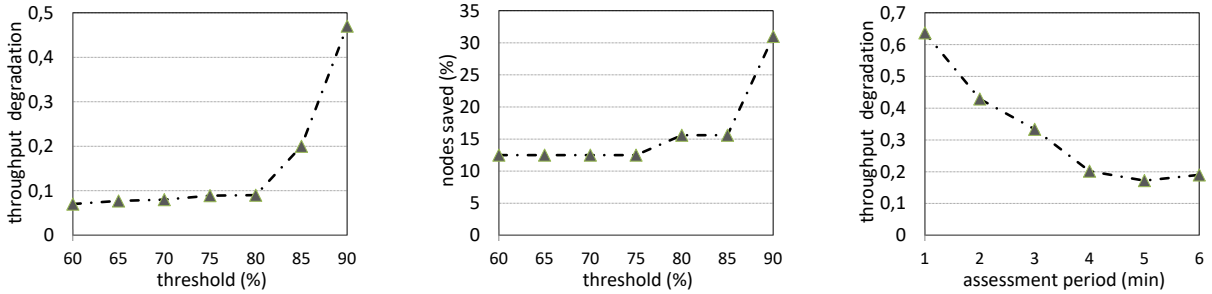
Evaluation Metrics

The effectiveness of ELYSIUM has been evaluated considering the following metrics:

- the ***throughput degradation***, measured as the percentage difference over time between input load and throughput, where the throughput is rate of acked input tuples (see Section 6.2). The throughput degradation is computed as $\frac{|input\_load - throughput|}{input\_load}$. Note that throughput degradation becomes greater than 1 when there is a large number of input tuples buffered in the queue. In this case the throughput can become much larger than the input load, hence $|input\_load - throughput| > input\_load$;
- the percentage of ***nodes saved*** with respect to a statically over-provisioned configuration; let N be the number of assessments done during the evaluation, C the number of worker nodes defined in the over-provisioned configuration, c_i the configuration chosen by the i-th assessment, this metric is computed as $1 - \frac{\sum_{i=1}^{N} c_i}{NC}$;

- the ***latency***, i.e. the average tuple completion time.

Whenever applicable these metrics have been computed over sliding time windows or as an overall value for the entire test.



(a) Throughput Degradation

(b) Nodes saved

(c) Overhead due to the assessment period

Figure 6.6: *Throughput Degradation (a,c) and nodes saved (b) due to parameters setup (threshold cpu_max_thr and assessment period)*

Parameters setup

To properly set the thresholds presented in Section 6.2, we adopted a methodology based on Reinforcement Learning. We used Q-Learning [230] during the profiling phase starting with no knowledge of the application behaviour.

To find the cpu_max_thr, the Reward Function $R(threshold)$ we propose aims at maximising node usage, hence looks for the maximum CPU threshold that corresponds to the lowest throughput degradation; specifically the reward function is defined as:

$$R(cpu\_max\_thr) = cpu\_max\_thr - throughput\_degradation$$

The Q-Learning rewards are shown in table 6.1 where it is possible to see that the max reward is given to a threshold of 0.8, i.e. 80% CPU usage. Figure 6.6 shows how the throughput degradation and nodes saved metrics change in function of the max CPU threshold.

Specifically, Figure 6.6(a) backups the result that the 80% of CPU usage seems to be the best cpu_max_thr as larger values impose a larger throughput degradation. In a similar way we computed the other thresholds: their values are 0.25 for $core\_min\_thr$ and 0.65 for $core\_max\_thr$.

The ANNs have been tuned by following some empirical rules presented in [239]: the ILP ANN has 13 input nodes, 1 hidden layer with 24 neurons, 5 output nodes for *direct prediction* (i.e. a prediction for each future minute) and linear-tanH-tanH activation functions.

Data are normalised with the min-max normalisation $[0; 1]$ and the dataset was split 70% training and 30% test. For OCUP/OP ANN we set 1 hidden layer with 3 neurons, *tanH-tanH-tanH* activation functions. We trained the ANNs with the Resilient Backpropagation [196] and a 10-cross validation to avoid overfitting.

CPU Max Threshold	Reward
0.60	0.53
0.65	0.57
0.70	0.62
0.75	0.66
0.80	**0.71**
0.85	0.65
0.90	0.43

Table 6.1: *Reward of Q-Learning for CPU* `max_threshold`

Stream	Average	Std. Dev.
`WordGener - StopWordFilter`	17.86	0.54
`StopWordFilter - Counter`	0.68	0.02
`Counter - IntermRanker`	0.41	0.34
`IntermRanker - FinalRanker`	0.01	0.00

Table 6.2: *Selectivity of T1's streams*

The profiling phase duration is application dependent. Basically, the more data you collect, the more accurate the prediction will be. In our scenario we notice that injecting a variable workload for 30 minutes is enough to achieve a good prediction accuracy (see next subsection).

6.5.2 ELYSIUM Evaluation

Reconfiguration Overhead

The overhead introduced by ELYSIUM is negligible. The metrics monitoring CPU usage and traffic are extremely lightweight (they are collected every 10 seconds).

The bandwidth consumed for metric collection is just few *KB*, depends on the number of operator instances, and it is independent from the input load. The real-time computation of the AutoScaler is lightweight and consumes an insignificant amount of CPU periodically. Furthermore, this computation is carried out on the machine hosting the Nimbus, so it doesn't compete for resources with running topologies.

When the configuration has to be changed, the throughput of an application degrades. In our experiments, a reconfiguration is triggered by issuing a rebalance command to the Nimbus, which causes such degradation for two reasons mainly: firstly, topologies have to be restarted ($R_{restart}$), which takes 5 to 8 seconds in our testbed.

During this period, the application cannot process tuples, so they are buffered before the spout component (into the JMS queue in our topologies). Secondly, once topologies become ready to

work, the spouts start retrieving tuples from the input queues at the highest possible rate. This is likely to causes a non-negligible load peak with a consequent resource overloading, regardless of the actual input load curve.

So, after the restart of the topology, a transient phase occurs where the cluster is likely to move in a stress state because applications need to drain accumulated input tuples (R_{queue}) to finally keep up with the real input load. The length of this transient phase depends on how many tuples are queued while the reconfiguration takes place. This is a common behaviour for SPSs that, like Storm, do not allow dynamic reconfigurations of running applications at runtime.

To measure how the assessment period impacts the reconfiguration overhead, we deployed T1 over an over-provisioned configuration (no worker nodes nor operators in stress state) and injected 9 minutes of sinusoidal input load. In this setting, we computed the throughput degradation for different assessment periods. As expected, Figure 6.6(c) clearly shows the throughput degradation gets larger as the assessment period is shortened.

By comparing these results with the quasi-zero throughput degradation obtained without reconfigurations and in an over-provisioned setting (see Figure 6.9), it can be noted that reconfiguration overhead is significant. Therefore, the assessment period has to be tuned accordingly to input load variability and throughput degradation tolerance. In our tests, we set the assessment period to 1 minute.

Therefore, pessimistically assuming reconfigurations occur at each assessment, the baseline value of the throughput degradation for comparisons is 0.64 (with 2 minute assessment period, the throughput degradation would be 0.43).

Estimator Accuracy

The accuracy of the estimations provided by the Estimator depends in turn on the accuracy of the profiles learned by the SP, the OCUP, and the OP. Table 6.2 shows average and standard deviation of the selectivities observed for the streams of T1, during a 30 minutes test with the stair-shaped curve as input load.

Reported standard deviations are very small, which backups the implementation choice for the SP of modeling selectivities with constant values. The stream *Counter - IntermediateRanker* is the only one having a large standard deviation. This is due to the semantics of the *Counter* bolt; indeed, it sends tuples downstream to the *IntermediateRanker* bolt periodically, independently of its input rate. The impact on the estimation is negligible as at runtime the input rate of the bolts downstream the *Counter* bolt is very small and produces really small CPU usage.

The accuracy of the OCUP is related to the estimations of CPU usage for an operator instance given its input rate. Average *mean percentage error* of estimations is under 3%. Figure 6.7 reports the real CPU usage over time of the instances of two operators, aggregated by operator, and the corresponding estimations provided by the OCUP. In this test, a sinusoidal input load was injected in the topology for 25 minutes. As the figure shows, the estimations faithfully predict the real CPU usage.

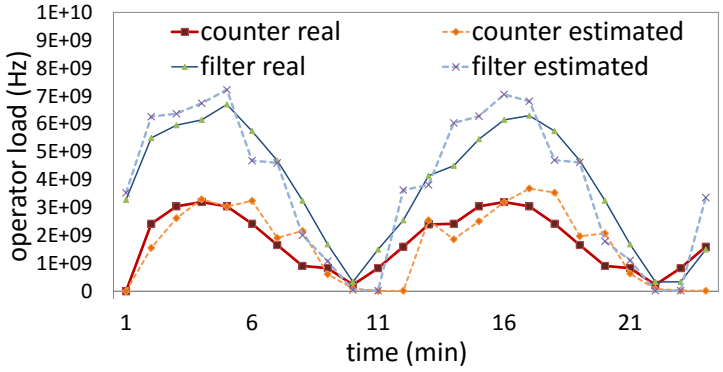


Figure 6.7: *Comparison between real and estimated total CPU usage (in Hz) for all instances of T1's Counter and StopWordFilter operators*

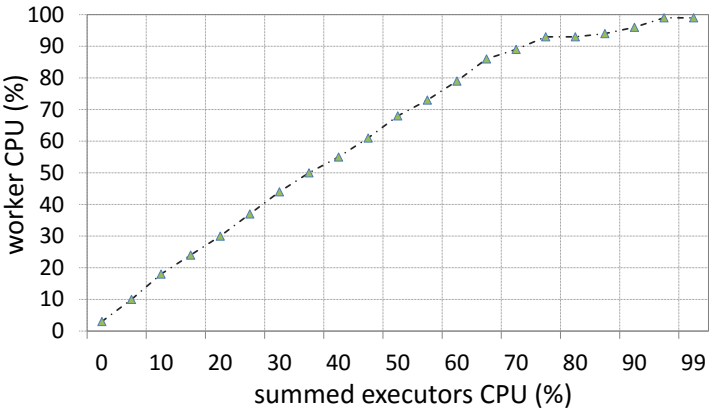


Figure 6.8: *Worker node CPU usage as a function of the sum of the CPU usage of all the executors running in such worker node*

The OP estimates the CPU usage of a worker node as a function of the sum of the CPU usage of all the operator instances running in that node. In this way, it is possible to take into account the overhead caused by the SPS such as tuple dispatching and thread management. Figure 6.8 depicts the profiling of such overhead in a worker node of our cluster. Such profiles provide all the information needed to infer the total CPU usage of a worker node.

Comparing Joint and Symbiotic Scaling

To define the policy enforced by the joint scaling approach, we took inspiration from [48][4]: *operator scale-out entails adding a new resource, while operators scale-in and resource scale-in/out are independent*. This means that another worker node is added whenever any operator is scaled-out, while operator scale-in doesn't affect resource scaling. Furthermore, in case no operator is scaled, resources are scaled in or out on the base of current worker nodes' CPU usage.

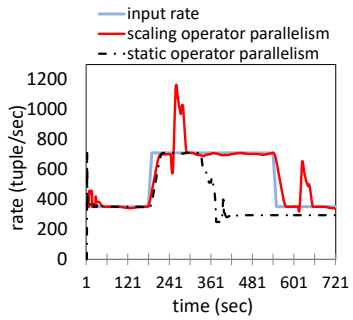
[4]Note that here we aim at comparing symbiotic versus joint approaches and not the systems themselves as they are widely different.

Topology / Dataset	Figures	Scaling Type	Resources	Operator Parallelism	Throughput Degradation	Nodes Saved %
T1 stair dataset	--	--	4	4	0.05	0
	--	only operators	2	scalable	1.47	50
	--	only operators	4	scalable	0.98	0
	--	only resources	scalable	2	1.43	19
	--	only resources	scalable	4	0.97	33
	10(f) 13(a)	joint	scalable		0.97	32
		ELYSIUM (R)	scalable		0.81	43
		ELYSIUM (P)	scalable		0.81	43
T1 step	**10(a-c)**	joint	scalable		0.78	50
		ELYSIUM (R)	scalable		0.59	58
T1 square	**10(d-e) 11(a) 13(b)**	joint	scalable		1.49	25
		ELYSIUM (R)	scalable		1.7	35
		ELYSIUM (P)	scalable		1.2	35
T1 sine	**10(g) 11(b)**	joint	scalable		1.25	25
		ELYSIUM (R)	scalable		1.01	47
T1 twitter	**10(h) 11(c) 13(c)**	joint	scalable		1.25	24
		ELYSIUM (R)	scalable		0.86	45
		ELYSIUM (P)	scalable		0.9	45
T2 square	**10(i) 11(d)**	joint	scalable		0.99	13
		ELYSIUM (R)	scalable		1.03	33
T2 sine	**10(j) 11(e)**	joint	scalable		0.63	30
		ELYSIUM (R)	scalable		0.17	22
T2 twitter	**10(k) 11(f)**	joint	scalable		0.49	48
		ELYSIUM (R)	scalable		0.48	50
T1 step T2 stair	**12(a-c)**	ELYSIUM (R)	scalable		0.80	17
T1 sine T2 sine	**12(d-f)**	ELYSIUM (R)	scalable		0.46	21

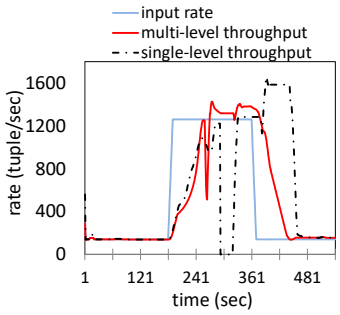
Figure 6.9: *Evaluation summary. The second column indicates the reference to the figure in this paper; the third column refers to the scaling strategy, where ELYSIUM can be set either reactive (R) or proactive (P).*

Since joint scaling is reactive, ELYSIUM was set in reactive mode as well and a same activation threshold for both approaches was considered, such to provide a fair comparison.

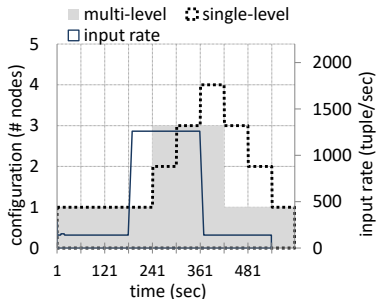
To highlight the advantage of scaling on a single dimension only, either operators or resources, we first show a case where scaling only operators, and not resources, can be enough to make the application sustain an input load peak. Figure 6.10(a) shows a throughput comparison over T1 between a static configuration and an operator-only AutoScaler. The static configuration has 2 worker nodes and all operators with parallelism set to 1, so there are 6 executors over 8 cores (2 worker nodes with 4 cores) running operators, and 2 remaining cores used by other Storm processes. The static configuration cannot sustain the input load change occurring at about second 180, and the throughput drops after a couple of minutes. The AutoScaler starts with the same configuration, then scales up when the peak occurs, as it detects an operator stress. It changes the parallelism of the stressed operator (*StopWordFilter* in this case) from 1 to 2 and the throughput,



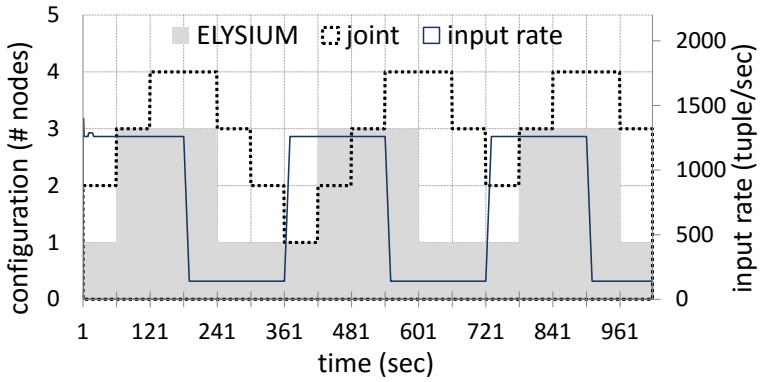
(a) Throughput with static vs operator scaling



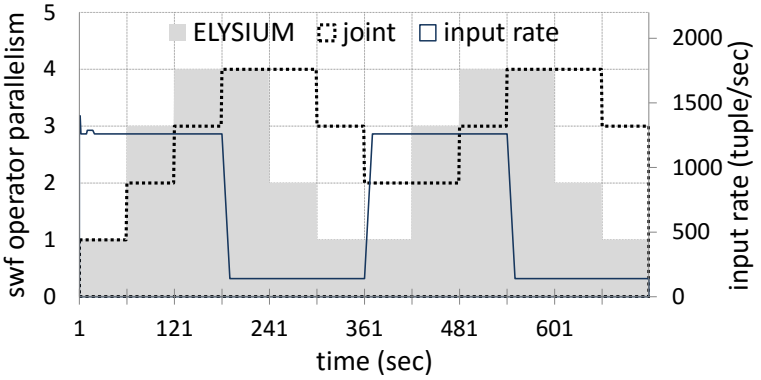
(b) Throughput with single vs multi-level scaling



(c) Nodes with single vs multi-level scaling

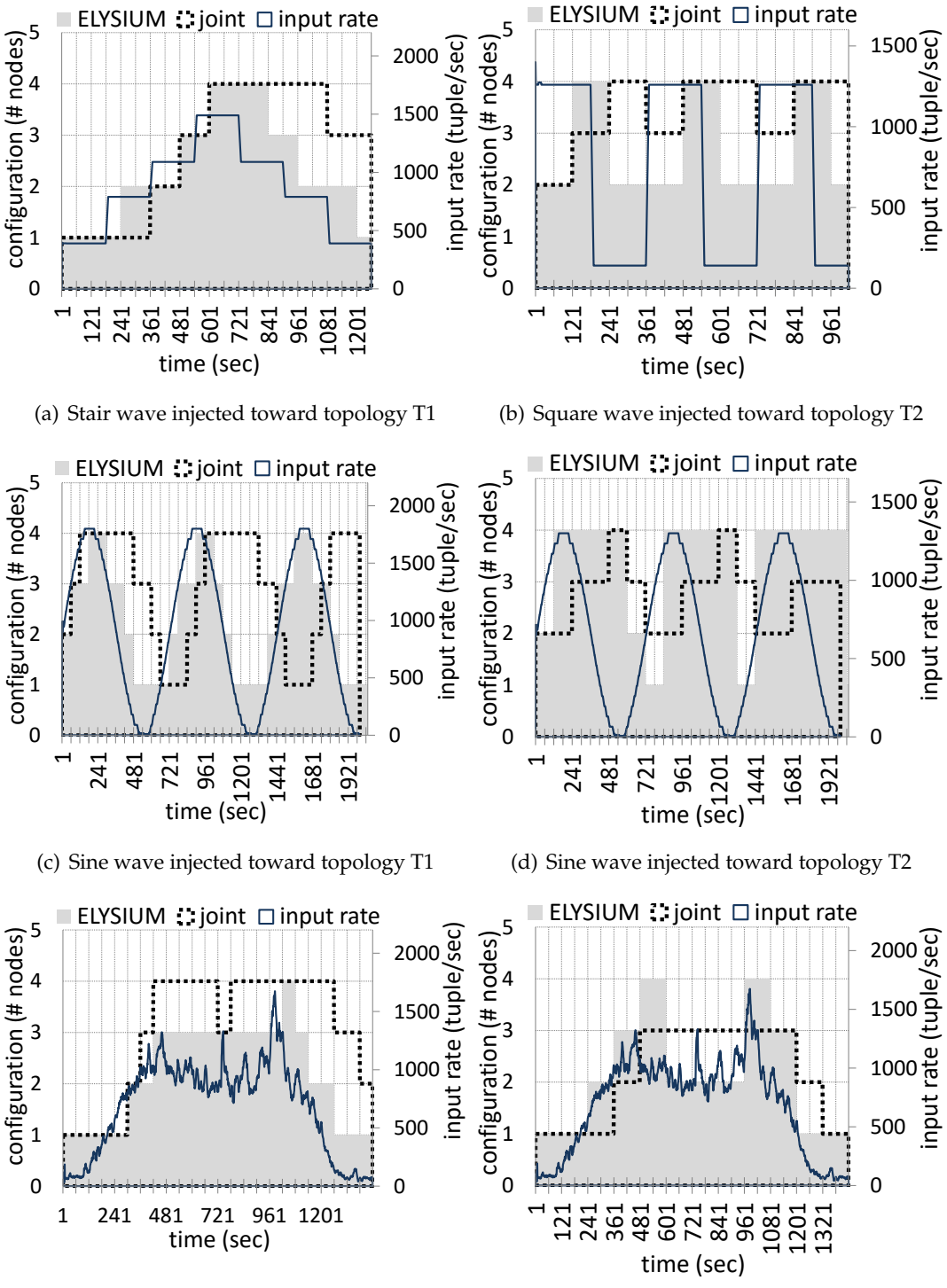


(d) Resources scaling while injecting a square wave toward T1



(e) Operators scaling while injecting a square wave toward T1

Figure 6.10: *Comparison between joint scaling and symbiotic scaling (ELYSIUM) while injecting different traces toward T1 and T2*



(a) Stair wave injected toward topology T1

(b) Square wave injected toward topology T2

(c) Sine wave injected toward topology T1

(d) Sine wave injected toward topology T2

(e) Twitter curve injected toward topology T1

(f) Twitter curve injected toward topology T2

Figure 6.11: *Comparison between joint scaling and symbiotic scaling (ELYSIUM) while injecting different traces toward T1 and T2*

after some oscillations due to reconfiguration overhead, increases keeping up with the input rate. The inverse operation (operator scale-in) occurs at about second 600, where the two operator instances become under-utilised and the parallelism is set back to 1.

The next experiment aims at underlining the limitation of the joint scaling regarding its possibility to scale resources in/out of a single unit (*single-level*). On the contrary, the proposed scaling approach leverages the Estimator to choose the proper number of worker nodes to use (*multi-level*). For this test, we used a step-shaped input load over T1, as shown in Figure 6.10(b) and 6.10(c), where throughput and used worker nodes comparisons are shown, respectively.

These Figures clearly show that both the scaling strategies suffer the input load peak at the beginning. While the symbiotic scaling resumes sustaining the input load after 80 to 90 seconds from the peak, the joint scaling makes the application throughput break down for a few tens of seconds, then manages to keep up after about two and half minutes from the input load peak. When the input load decreases at minute 6, the symbiotic approach scales in the resources after one minute, and the throughput gradually decreases to match the input load. During this settlement period, the throughput is larger than the input load because of the reconfiguration overhead.

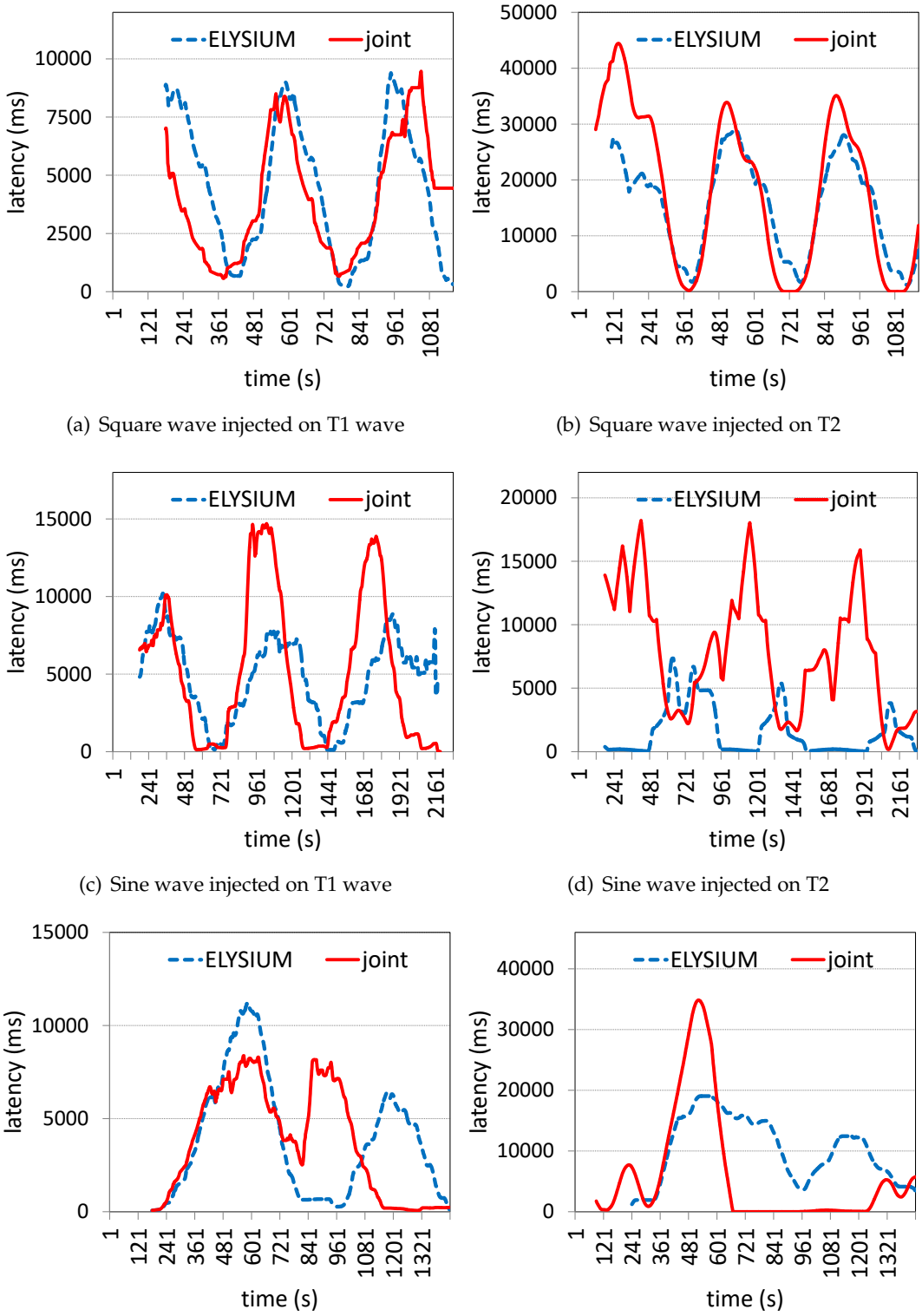
The joint scaling performs worse as it requires more reconfigurations to reach the correct one, so it pays a much larger overhead, while ELYSIUM provisions the right amount of resources with a single reconfiguration. Indeed, joint scaling takes a few tens of seconds longer than ELYSIUM to generate a throughput equal to the input load.

Besides providing smaller throughput degradation (0.59 against 0.78), the symbiotic approach allows to save resources as show in Figure 6.10(c) (see also Figure 6.9, where an overview of all the tests executed is reported). Throughput degradation of symbiotic scaling is slightly better than that obtained by reconfiguring every minute (see Figure 6.6(c)).

To provide a better understanding on the way operators and resources are scaled symbiotically, we show the results of an experiment that used a square wave input load over T1. Figure 6.10(d) and 6.10(e) present respectively how the number of worker nodes and the parallelism of T1's StopWordFilter (the most significant operator in T1) change over time, for joint and symbiotic scaling (i.e. ELYSIUM).

With the symbiotic approach it is possible to adapt faster than with the joint one, for what regards both the resources and the operator parallelism. The throughput degradation is similar but larger with ELYSIUM (1.7 vs 1.49) as a lower number of nodes is used compared to the joint approach, which instead over-provisions the topology and does not experience overloading. Indeed, nodes saved are 25% for joint scaling and 35% for ELYSIUM. We experienced similar results with T2 (Figure 6.11(b)): slightly larger throughput degradation (1.03 vs 0.99), but more nodes saved (33% vs 13%).

To complete the comparison between joint scaling and ELYSIUM, we show how they differ in used worker nodes over time for other distinct input load curves. Figure 6.11(a) shows the comparison with a stair-shaped input load over T1. Globally the throughput degradation is smaller for



(a) Square wave injected on T1 wave

(b) Square wave injected on T2

(c) Sine wave injected on T1 wave

(d) Sine wave injected on T2

(e) Twitter wave injected on T1

(f) Twitter wave injected on T2

Figure 6.12: *Comparison on latency between ELYSIUM and joint scaling while injecting different input load curves toward the two topologies*

ELYSIUM (0.81 vs 0.97 of the joint), while saving more resources (43% vs. 32% with joint scaling).

Similar results are reported in Figure 6.11(c) and 6.11(d) for a sinusoidal input load over T1 and T2 respectively. In both cases, ELYSIUM provides a lower throughput degradation (1.01/0.17 vs 1.25/0.63), while they differently save nodes (47/22% vs 25/30%).

Finally, Figure 6.11(e) and 6.11(f) show the results with the Twitter trace over T1 and T2. In both tests ELYSIUM has a lower throughput degradation (0.86/0.48 vs 1.25/0.49) and more nodes saved (45/52% vs 24/48%).

The performance of ELYSIUM compared to joint scaling in terms of latency are shown in Figure 6.12. Specifically, it is possible to see that the trend of the latency of both approaches when injecting a square curve is similar for both T1 and T2 (Figure 6.12(a), 6.12(b)). For the twitter trace, instead, ELYSIUM and joint scaling, for both T1 and T2, differ in specific periods as they use a different policy to scale (Figure 6.12(e), 6.12(f)).

The main differences are appreciable from tests using a sinusoidal wave as input load, as shown in Figure 6.12(c), 6.12(d) where ELYSIUM outperforms the joint approach, showing pretty smaller latency values.

Managing Multiple Applications

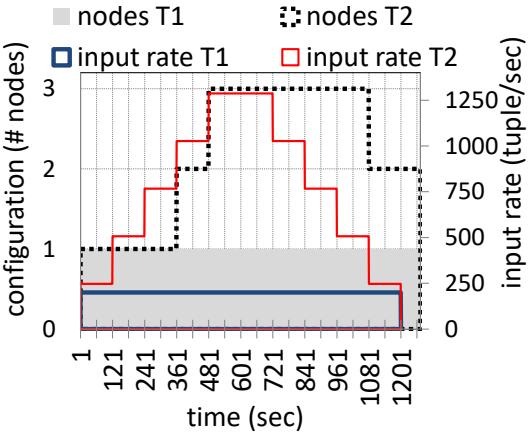
To test the ability of ELYSIUM to scale in presence of multiple applications, we ran two tests with both T1 and T2 deployed in the same cluster. In the first test we injected a step input load of $200 req/s$ in T1 and a stair wave in T2.

From Figure 6.13(a,b) it is possible to see how the two topologies require different number of nodes as well as different number of operators. Specifically, the nodes of T2 change over time as it has to handle a larger workload, while T1 always uses a single node. Nevertheless, T1 frequently requires an increases of its operator parallelism, as the overhead due to reconfigurations leads to a larger usage of some operators.

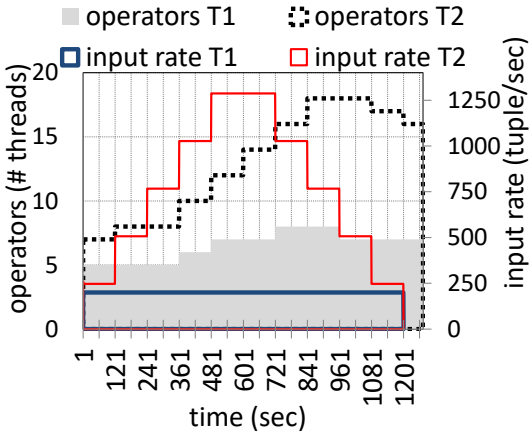
Conversely, in a second test we injected a sinusoidal input load in both T1 and T2 with different magnitudes. Figure 6.13(c,d) show how T1 and T2 differently scale for nodes and operators. From Figure 6.13(e,f), it is possible to see how the latency of both T1 and T2 is quite stable and, obviously, T2 has in both cases larger values as it handles a larger workload.

Proactive Symbiotic Scaling

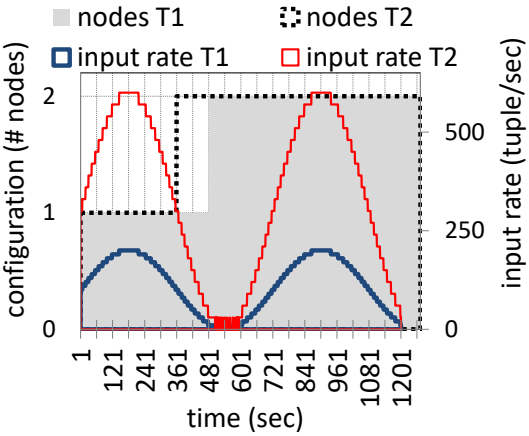
ELYSIUM can be used in either reactive or proactive mode. Proactive scaling can help reducing the delay between when the reconfiguration occurs and when its effects are actually needed. Here, we implemented a technique that over-provisions resources for the next temporal horizon. By setting for instance a horizon of h minutes, the proactive system computes a prediction of the input load for each minute from $t+1$ to $t+h$, and uses the highest forecasted input load to estimate required resources.



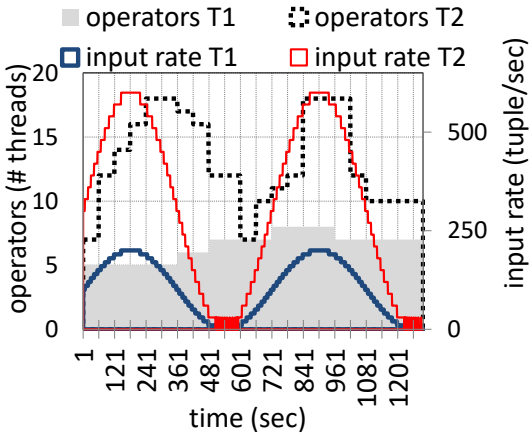
(a) Nodes used by T1 and T2



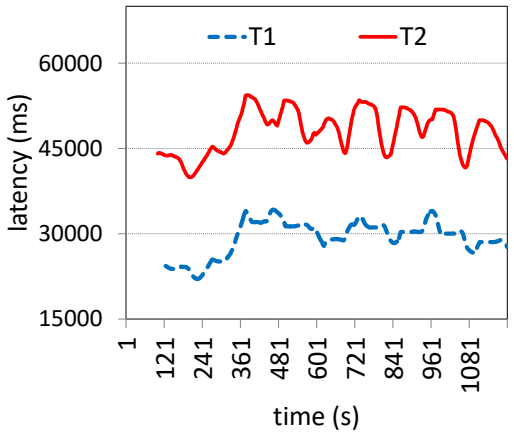
(b) Operators used by T1 and T2



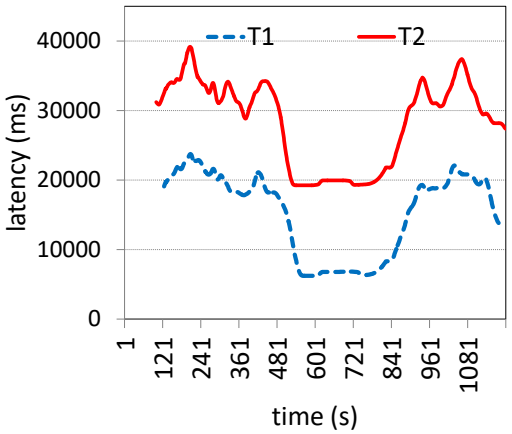
(c) Nodes used by T1 and T2



(d) Operators used by T1 and T2 trace

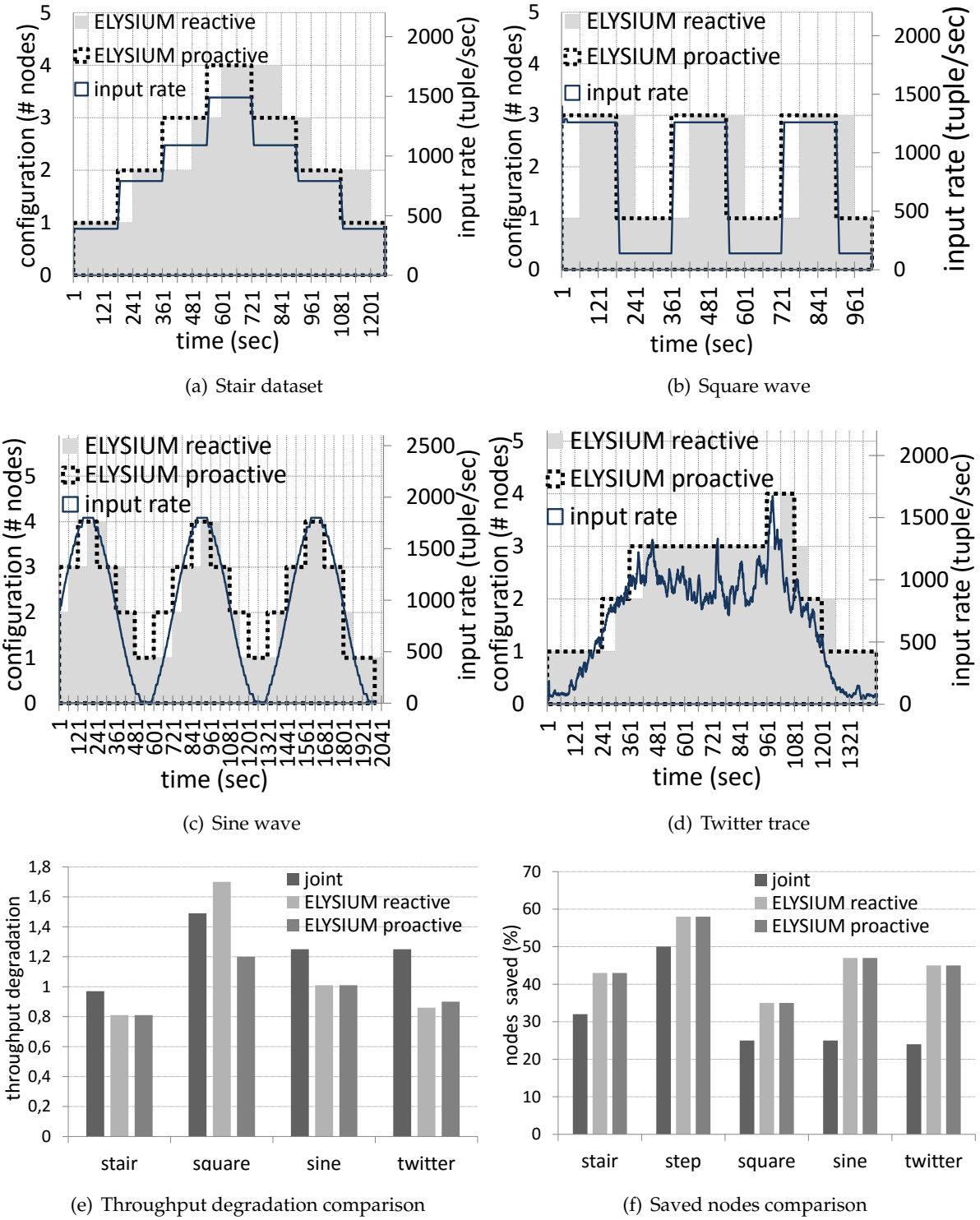


(e) Latency of T1 and T2



(f) Latency of T1 and T2

Figure 6.13: *ELYSIUM handling two topologies with different workloads*



(a) Stair dataset

(b) Square wave

(c) Sine wave

(d) Twitter trace

(e) Throughput degradation comparison

(f) Saved nodes comparison

Figure 6.14: *The Figures above show the comparison on used worker nodes between reactive and proactive ELYSIUM according to specific input load curves toward T1. The Figures below show the comparison on throughput degradation and nodes saved aggregates between joint and symbiotic reactive/proactive approaches, with different input load curves.*

Figure 6.14 (a-c) show the comparison on used worker nodes between reactive and proactive ELYSIUM, while injecting three different input load curves toward T1.

Note that the strategies used by the proactive and reactive systems are exactly the same, the unique difference lying in the reconfiguration point that for the proactive version results closer to the real demand point.

In terms of nodes saved, the differences are negligible (very few nodes), but available resources are used more efficiently. Figure 6.14 (d-e) shows the overall results of these comparisons in terms of throughput degradation and nodes saved. For the square wave input load (i.e. the most critical pattern for reactive ELYSIUM), the throughput degradation drops from 1.7 to 1.2 showing a notable improvement that clearly justifies the usage of a proactive approach.

6.5.3 Overall Result

The overall main results are: (i) ELYSIUM always outperforms the joint scaling approach in term of saved resources, (ii) ELYSIUM is anyway able to sustain the same workload, often with a lower throughput degradation and lower latencies due to its ability to scale more units per time and (iii) the proactive version of ELYSIUM can reduce the impact of the reconfiguration overhead and further improve performance of the symbiotic approach.

6.6 Discussion

In this chapter we presented ELYSIUM, an elastic scaling solution for SPS that instantiates the PASCAL architecture. The PASCAL's profiler is implemented with a set of submodules to learn the behaviour of a SPS application, then predictively scales the system symbiotically along two distinct dimensions: operator parallelism and resources. Through an experimental evaluation based on a real prototype integrated in Storm, we showed how ELYSIUM outperforms a joint scaling strategy, while always saving more resources.

The SPS scenario has been particularly interesting as it needs to consider two dimension along which to scale. An important extension may regards a more complete model for resource estimation including memory and bandwidth so as to integrate shedding techniques to tackle bandwidth bottlenecks. Moreover, by considering even other optimisation techniques proposed in [116], we also plan to integrate further solutions (e.g. smart operator placement to improve load balancing among resources) and scaling according to predefined SLAs, such as maximum latency, as we similarly proposed in MYSE and in the PASCAL scenario for distributed storage.

Part II

Blockchain-based Architectures

Chapter 7

Leveraging Blockchain for Secure Multy-party Interactions

Elasticity is one of the most prominence properties of Cloud computing, as providers allow to use resources on-demand with a pay-as-you-go price model. As assessed in the Part I of this thesis, automatic scaling systems allow to elastically modify a configuration to achieve high performance within a desiderata QoS, while reducing costs, by employing only necessary resources.

Recently, Cloud providers, in order to increase their elasticity in providing resources to users, started investigating the concept of Cloud federation [80, 205, 217]. A federation of Cloud may improve the elasticity of a provider which can "lend" extra unused resources to other federation members or "borrow" them if needed. A new form of elasticity may so replace *automatic scaling* solutions with *resources sharing* among federation members.

Besides, the concept of Cloud federation is fostering also public and private companies which aim to carry out interoperability through a cooperation network. Thus, they are prompting a resource sharing of their already deployed private cloud systems (see, e.g., the ENISA report in [80]) so as to ease multi-party interactions, and enabling so goal-oriented federations.

Although providers and companies can benefit for resource sharing and for a design of a common infrastructure, they need a clear understanding of the potential of each federation decision, so as to choose the most effective solution depending on the environment conditions. Different works, studies the economics of Cloud federation to enhance providers' profit as [97].

Besides the multiple technical issues to address, the creation and management of cloud federations have to face daunting security issues, mainly related to (i) a lack of trust among federated partners, (ii) the non-disclosure of sensitive data and (iii) the enforcement of integrity guarantees.

In this part of the thesis, we investigate blockchain-base architectures to securely federate multiple parties through a common infrastructure balancing performance and security. As a reference case study we consider the EU SUNFISH project, which aims to realise a secure Cloud federation. In this chapter, specifically, we propose design and implementation of the underlying infrastruc-

ture of SUNFISH, namely Federated Service Ledger. We then show how to employ such an infrastructure to use a permissioned blockchain for a SUNFISH use case related to a secure payslip computation involving the Italian Ministry of Economy and Finance and the Ministry of Interior.

Contributions. The content of this chapter related to the SUNFISH project are available online at the GitHub repository [193] and at the official documentation [71]. Furthermore, the content of Section 7.3 has been published in [179].

Chapter Structure. In Section 7.1 we introduce the background on blockchain technology, detailing main differences between permissionless and permissioned settings and introducing the role of distributed consensus in such systems. Then in Section 7.2 we introduce the SUNFISH project as a solution to create multi-party federation, focusing on design and implementation of the underlying Federated Service Ledger infrastructure. Finally, in section 7.3, we propose a solution based on the aforementioned Federated Service for the secure payslip computation use case.

7.1 Blockchain Technology

The blockchain is a quite novel technology that has appeared on the market in the recent years, firstly used as public ledger for the Bitcoin cryptocurrency [174]. It mainly consists of consecutive chained blocks containing records, that are replicated on the nodes of a p2p network.

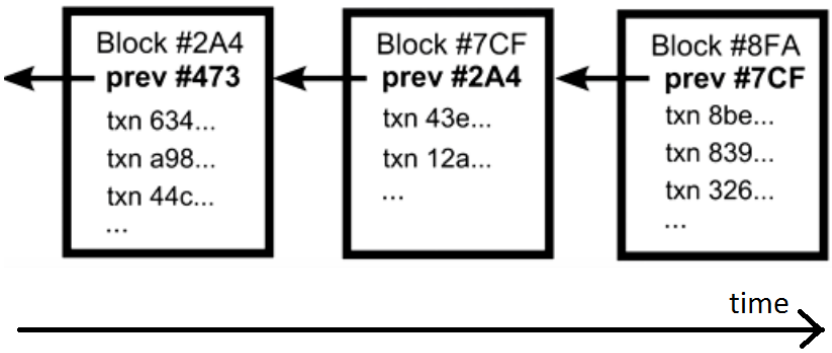


Figure 7.1: *Blockchain representation with blocks referencing each other through the hash of the previous*

These records witness transactions occurred between pseudonyms. Transactions may feature a cryptocurrency like, e.g., the Bitcoin, or other kinds of assets. Blockchain also offers so-called *smart contracts*, immutable programs deployed and executed on top of a blockchain system, e.g. Ethereum [234], to realise decentralised applications where no involved party is in control neither of the code nor of the data. The collection of transactions and their enclosing in chain blocks is carried out in a decentralised fashion by distinguished nodes of the network, i.e. *miners*. Miners

apply opportune block construction methods, i.e., the *mining process*, to achieve consensus among all the miners on newly generated blocks. Bitcoin is an example of *permissionless* blockchain, i.e., there is no restriction for a node to become a miner. If instead there is an authentication and authorization layer for miners, then the blockchain is *permissioned*. A complete work which compares the two approaches has been proposed by BitFury and Garzik in [100] and [101].

7.1.1 Permissionless Blockchain and Proof-of-Work Mining

The original mining process, still used for Bitcon and Ethereum blockchain, is based on the *proof of work* (PoW). It consists in a computational intensive hashing task. Specifically, to solve a block a miner needs to find a random number (also called *guess*) which acts as a proof of work (see Figure 7.2); indeed, such a number concatenated to transactions collected inside the block has to provide the overall hash of the block less than the target number. The target number is regulated according to the so-called *blockchain difficulty* that regulates the average time spent by miners to accomplish such a task and create a new block.

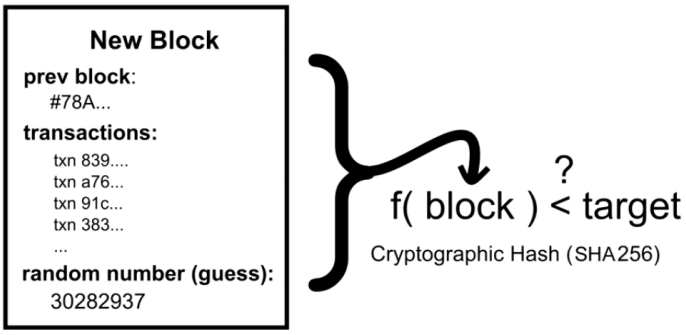


Figure 7.2: *Proof-of-Work as a computational puzzle to solve a block*

Once a miner achieves the creation of a new block, it broadcasts that block to all the other miners. They consider such a block as the latest of the chain and start mining new blocks to be appended. For the sake of simplicity, we can say that once a miner has created a new block, it becomes part of the chain. However if multiple miners concurrently add a block, a transient fork is created which is usually resolved over time by concatenating other blocks as by design miners always consider the longest chain. Miners' incentive to correctly support the network is due to rewards obtained for successfully mining a block and for fees that each transaction can optionally acknowledge to the miner [174, 99].

PoW-based blockchains enjoy many fascinating properties related to *data integrity*, which follow from the mining process and from the full replication of the blockchain on a large number of nodes. Indeed, when a block is part of the chain, all miners have agreed on its contents, hence it is practically non-repudiable and persistent unless an attacker has the majority of miners' hash power that are able to create a fork of the chain. Assuming a majority of hash power controlled by honest miners, the probability of a fork of depth n is $\mathcal{O}(2^{-n})$ [36]. Since the probability of a

forks decreases exponentially, the users simply waiting for a small number of blocks to be added (e.g. 6 in Bitcoin) can be sure their transactions are permanently included with high confidence. This allows to avoid the *double spending attack*, i.e. a successful spending of some money more than once [221]. However, in [82] the authors showed as "*majority is not enough: Bitcoin mining is vulnerable*" if only 25% of the computing power is controlled by an adversary.

Although such blockchains provide strong integrity properties, PoW-based blockchains have a main drawback: *performance*. This lack of performance is mainly due to the broadcasting latency of blocks on the network and the time-intensive task of PoW. This is due to the fact that thousands of miners spread world-wide are required to render tampering with transactions computational infeasible and broadcasting blocks over this kind of network topology takes very long. Thus as a matter of fact, each transaction stored on a blockchain has a high confirmation latency, which causes an extremely low transaction throughput. In Bitcoin, the average latency is 10 minutes, and the throughput is about 7 transactions per second, while for Ethereum the latency is on average 10 seconds with 15 transactions per second as throughput [36].

Another relevant concern related to the use of the blockchain regards its *stability*. Although, e.g., the Bitcoin's blockchain has worked quite well so far, there is no universally accepted academic work explaining either why this has happened, or whether it will continue in the future, or how long it will [36]. The stability properties of the PoW-based consensus protocol are still being debated, and current "literature does not even provide adequate tools to assess under which economic and social assumptions Bitcoin itself will remain stable" [36]. In general, PoW-based blockchains using incentive mechanisms based on cryptocurrencies are heavily subjected to market fluctuations, which casts a shadow on the blockchain effectiveness on the long term.

Finally, PoW is energetically inefficient leading a huge waste of money and resources to make a computation. Thus, as alternative consensus algorithm has been proposed the Proof-of-Stake (PoS) with the Nxt cryptocurrency [57]. PoS employs a deterministic (pseudo-random) solution to define the creator of the next block and the chance that an account is chosen depends on its wealth (i.e. the stake). In PoS the blocks are said to be *minted*, rather than mined. Moreover, generally all coins are created at the beginning, and thus the total number of coins never changes afterwards, but there exist some PoS versions where new coins can be created. Therefore, in the original version of the PoS there are any block rewards, hence the minter take only the transaction fees [189].

7.1.2 Permissioned Blockchain and Consensus Algorithms

Conversely to permissionless blockchains, permissioned blockchains usually do not employ cryptocurrencies and their purpose is more oriented to exchange of asset and transactions. They can be faster by design, as the authentication of the nodes in the network changes the trust level, allowing thus to employ a consensus schema lighter than the PoW. A report of Swanson compares the two models [219]. Different consensus algorithms have been proposed for permissioned blockchain [42] each one having different requirements and guarantees.

Security of consensus algorithms is evaluated on two main properties [103]:

> ***Safety.*** *This property states that the algorithm should not do anything wrong during its normal execution. This property prevent unwanted executions and ensure different properties like consistency, validity agreement, and so on. If it is violated at some point in time, safety property will be never satisfied again.*
>
> ***Liveness.*** *This property ensures that eventually something good happens. It is a property of a distributed system which grant that the algorithm works properly in time and that sooner or later every execution completes correctly.*

For blockchain, consensus means achieving a global order on transaction ordering; in distributed systems jargon, this property is commonly known as *atomic broadcast* or *total order broadcast*. Atomic broadcast is a *reliable broadcast* which ensures also that each correct node outputs or *delivers* the same sequence of messages. Specifically it ensures the following properties [103]:

> ***Validity.*** *If a correct node p broadcasts a message m, then p eventually delivers m.*
>
> ***Agreement.*** *If a message m is delivered by some correct node, then m is eventually delivered by every correct node.*
>
> ***Integrity.*** *No correct node delivers the same message more than once; moreover, if a correct node delivers a message m and the sender p of m is correct, then m was previously broadcast by p.*
>
> ***Total Order.*** *For messages m_1 and m_2, suppose p and q are two correct nodes that deliver m_1 and m_2. Then p delivers m_1 before m_2 if and only if q delivers m_1 before m_2.*

According to the type of algorithm, forks can be possible. A further property excludes forks, i.e. the *Finality*. Specifically from [229]:

> ***Finality.*** *Given two correct nodes n_1, n_2, if n_1 appends a block b to its copy of the blockchain before appending block b', with $b \neq b'$ then no correct node n_2 appends block b' before b to its copy of the blockchain.*

Although the reference model is the State Machine Replication (SMR) [206], a fair comparison between each algorithm is still considered challenging [42] as some of them requires a *synchronous* network, while others can work in a *eventually synchronous* network which may strongly impact on performance and scalability. Note that a protocol may stall during asynchronous periods and it cannot be avoided due to a fundamental discovery by Fischer et al., i.e. the celebrated "FLP impossibility result" [84], which rules out that deterministic protocols reach consensus in fully asynchronous networks. Thus, the most accepted network model relies on *eventual synchrony* proposed in [74].

All those algorithms can be categorised in two families: (i) *Crash Tolerant* and (ii) *Byzantine Fault Tolerant* (BFT), i.e. able to resist to some subverted (malicious) node which can behave arbitrarily by sending also fake messages [145].

The main references for those families are (i) Paxos [143, 144], proposed by Lamport in its original version only tolerant to a crash of $t < n/2$ nodes, and (ii) Practical Byzantine Fault Tolerance (PBFT), proposed by Castro and Liskov [47], which is able to resist to $f < n/3$ Byzantine nodes.

The PoW algorithm, even though is not related to neither families, can be categorised as a BFT algorithm on a SMR model without *finality* guarantees. Indeed, it probabilistically achieves consensus on block (and thus, on transaction) ordering by voting through its computational power with possibility of (temporary) forks. A formal equivalence between the task solved by the "Nakamoto protocol" inside Bitcoin and the consensus problem in distributed computing is well presented in the work of Garay et al. [90]. A complete comparison between PoW and BFT algorithms has been proposed by Vukolic in [229].

An exhaustive comparison between different algorithms employed by existing technologies among those properties is available in [42]. The work compares the most important blockchain technologies such as (i) Hyperledger Fabric [83] with Apache Kafka [132] (i.e. a crash tolerant pub/sub) and with SIEVE (a protocol which combines PBFT and Eve [133]), (ii) Tendermint [140] a BFT consensus protocol for blockcahin similar to PBFT, (iii) R3 Corda with Raft [183] (i.e. a popular variant of Paxos, thus crash tolerant) and BFT-SMaRT [30] (i.e. the most advanced and tested implementation of a BFT consensus), (iv) Multichain [172], (v) Sawtooth Lake [141] with the Proof-of-Elapsed-Time (a.k.a. PoET, i.e. a novel protocol based on the insight that PoW essentially imposes a mandatory but random waiting time for leader election) and many others.

7.2 Case Study: the EU SUNFISH Project

The EU SUNFISH project[1] aims at proposing a distributed, democratic cloud federation platform that will ensure by-design the security of the managed data.

The SUNFISH proposal is *Federation-as-a-Service* (FaaS) [205], a new and innovative service that enables the secure creation and management of cloud data and services. FaaS features advanced data security services and innovative design principles leading to a distributed and democratic cloud federation governance. Anyway, they are out of scope for this thesis and further details can be found in [217, 232].

Members of such a cloud federation provide each other access to some of their own services, which can be consumed in the form of inter-cloud interactions. Figure 7.3 shows the resource usage of the Italian Ministry of Economy and Finance (MEF) –which is one of the partner of a SUNFISH cloud federation– for the salary processing. It is possible to note as the red area, i.e. used resources, is much lower than the blue area, i.e. available resources, which can be so shared

[1] `http://www.sunfishproject.eu/`

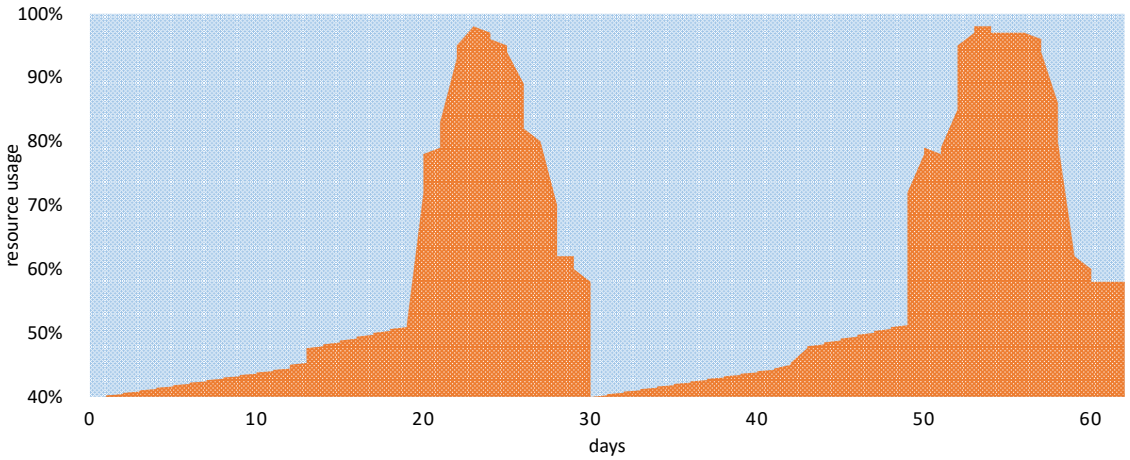


Figure 7.3: *Resource usage (red area) of the Italian Ministry of Economy and Finance*

among other federation members. Figure 7.4 shows the interaction within the SUNFISH federation between two of the members, i.e. MEF and MIN (Ministry of Interior).

The rules governing these interactions, hence the service usage, are defined in specific contracts. For instance, a member providing a service may require that only specific consumers can use it and that the service outputs have to be masked for privacy reasons. Due to the high sensitivity of the data managed by cloud federations (e.g., personal and medical data in case of the public sector), FaaS must provide high assurances about the compliance of the member contracts. Indeed, besides the runtime enforcing of the contracts, FaaS has to guarantee the integrity of contracts, namely that they cannot be tampered with and that all involved members must be aware of their existence. Additionally, to ensure non-repudiable evidences of contract enforcement, all the inter-cloud interactions have to be monitored and the logs stored with strong integrity guarantees.

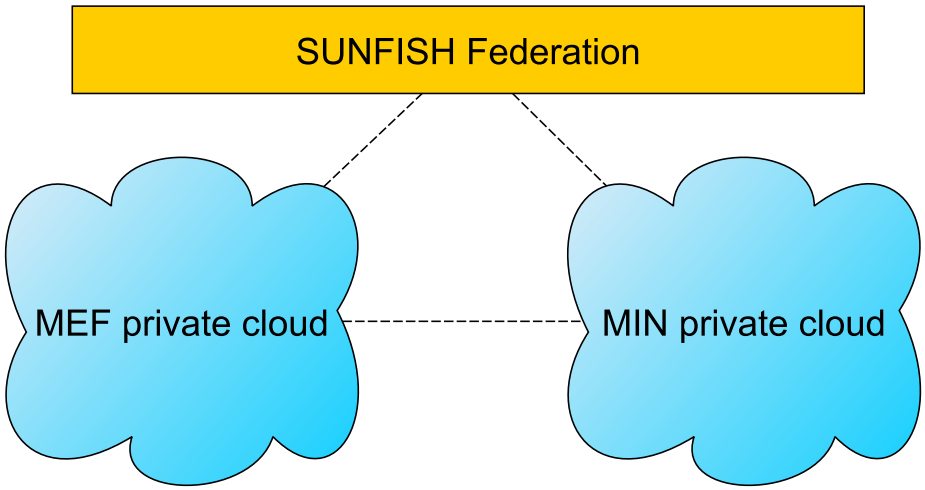


Figure 7.4: *MEF and MIN interaction within the SUNFISH federation*

Most of all, to foster a wide adoption of cloud federations, FaaS advocates the absence of a centralised governance. As a matter of fact, among federation members there cannot be designed a leader (i.e. there is no *primus inter pares*), rather federation members form a network of peers. To this aim, FaaS seeks to establish a decentralised, democratic federation governance, hence it must rely on an opportunely defined, distributed database ensuring strong integrity guarantees. The novel design solution for FaaS advocated by the SUNFISH project is based on the exploitation of a blockchain.

To properly address the feasibility of such a solution, significant threats to data integrity have to be identified as well as performance requirements. Specifically, current throughput and response time provided by typical blockchain technologies are not good enough for SUNFISH, furthermore we need a solution to cope with malicious member in the federation.

7.2.1 Federated Service Ledger Infrastructure

My main contribution on the SUNFISH project has been design and implementation of the underlying infrastructure interacting with the blockchain. Specifically, I worked on the *Federated Service Ledger* infrastructure that is composed by three main logical subsystems:

- Computing/Storage Platform
- Service Ledger
- Service Ledger Interface

The *Computing/Storage Platform* is the low-level underlying infrastructure to execute computations and store/retrieve values. It is assumed to be distributed on each federation member composing the infrastructure tenant and it can be implemented either with just a datastore or a blockchain. Computations in a decentralised fashion are anyway possible only by employing a smart contract enabled blockchain. A datastore or a pure blockchain platform (aka à la Bitcoin) can instead just store values.

The *Service Ledger* (SL) is a distributed stateless REST API server operating as logical interface of the Computing/Storage Platform. As the underlying computing/storage platform, it is deployed on each federation member of the infrastructure tenant. The Service Ledger enables the communication with the Computing/Storage Platform. It exposes three main operations:

- `get(k)`
- `put(k,v)`
- `invoke(args)`

The operations `get(k)` and `post(k,v)` can be used with both a datastore and a blockchain as long as the memory model is based on a key-value store (KVS). Thus, NoSQL-like datastore can be employed as well as most common blockchain as they directly rely on a KVS (with versioning).

The input parameters are a key `k` and an associated value `v`. The `get(k)` operation returns the associated value `v`, while the `put(k,v)` returns confirmation of an insert of the pair key `k` and value `v`. In case of smart contract-based blockchain as underlying platform, the `invoke(args)` operation can be used to invoke a computation on a smart contract and store the results on the blockchain. The REST API specification is available as yaml file on the Service Ledger section of the SUNFISH Platform API GitHub repository[2] and commented as part of the Platform documentation [71].

Finally, the *Service Ledger Interface* (SLI) is a stateless web app used as entry point to issue computations towards the SL outside the infrastructure tenant. Each component of the federation runs an instance of the SLI which as many operations as deployed services (or smart contract) running on the underlying Computation/Storage Platform. The SLI converts the received input in a couple key-value or in an 'args' format and, consequently, invokes the corresponding API of the SL. The full API specification is available in 2. By splitting the overall Federated Service Ledger in the just presented three modules permits:

- increasing flexibility, as multiple underlying infrastructure can be easily plugged-in: they just need to implement the three API put, get and invoke;
- increasing modularity, as interaction between components and the low-level infrastructure occurs via two levels of API.

Therefore, platform components, being within or outside the infrastructure tenant, can access all Service Ledger service via the exposed API considering the underlying infrastructure as a black box. Moreover, being multiple SLI/SL acting as entry points for the platform, the availability of the platform itself is strengthen. Notably, the Service Ledger infrastructure offers computational means not just for the federation governance, but also to empower cross-Cloud services. In fact, via a high-level SLI API the SL API invoke can be used to move part of the computation on, e.g., a blockchain smart contract. This gives the benefit of decentralised, immutable and non-repudiable computation. By way of example, if two member clouds need to share a decentralised computation they can use an application-oriented smart contract deployed on the blockchain via such high-level API. This is indeed the case of the payslip use case that we describe in Section 7.3 where part of the computation is moved to blockchain.

Figure 7.5 shows the high-level architecture of the Federated Service Ledger. The infrastructure tenant, created across all the member clouds, features one or more SL instances (yellow blocks below) to connect with different underlying platforms; we consider three potential platforms: a datastore, a blockchain and a smart contract enabled blockchain. It comes without saying that in case of distributed infrastructure, say blockchain, such infrastructure has to be the same. The SL can be invoked by an instance of the SLI (the blue blocks below) from any tenant, that in turn can be invoked by any component. The components part of the tenants can then use the SLI API as per the interaction (note that description of such components is out of the scope of this part of the thesis, that is instead focussed on the underlying infrastructure).

[2]https://github.com/sunfish-prj/SUNFISH-Platform-API/blob/master/ServiceLedger/

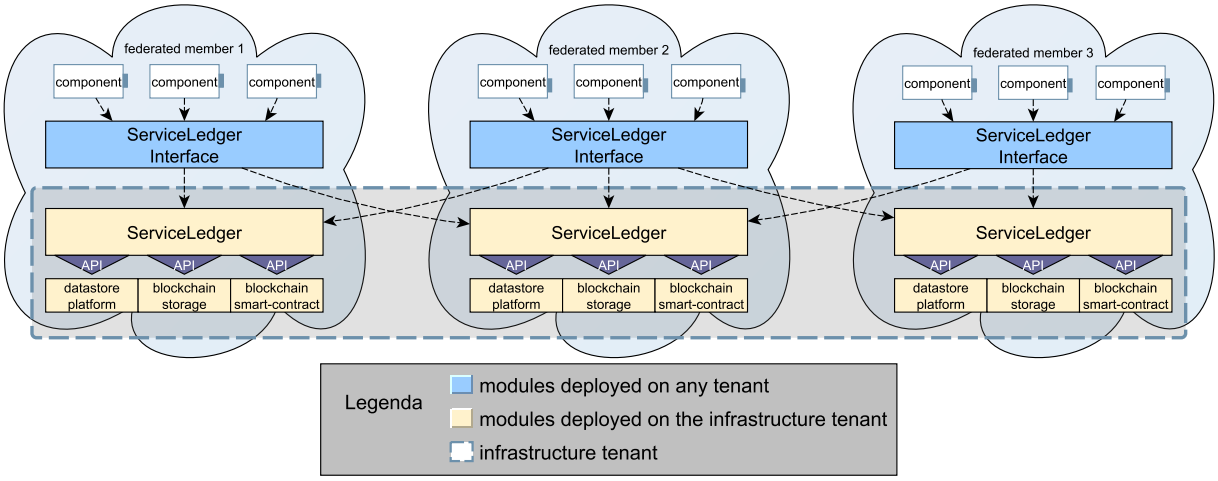


Figure 7.5: *High Level Architecture of Federated Service Ledger within the SUNFISH federation*

7.2.2 Federated Service Ledger Implementation

The component SL and SLI are NodeJS [181] web-services offering RESTful API. The API definition of the components have been designed via the Swagger 2.0 framework [218] which ensures parsing and sanitising of all JSON input/output. Both SL and SLI implementation code is available on GitHub [148, 125] and their installation guide available on the official guide [71]. The integration of SL with the underlying infrastructure is guaranteed for:

- MongoDB 3.4.x [169] as no-SQL data-store;
- Hyperledger Fabric 1.0 [83] as smart contract blockchain for permissioned systems.

As reported in Figure 7.6, the Federated Service Ledger modules have been deployed over the Clouds of three SUNFISH partners, i.e. Malta Information Technology Agency (MITA), MEF and MIN. In their section of the overall infrastructure tenant there are an instance of the SL web app, an instance of MongoDB and Hyperledger Fabric. More specifically, the Fabric blockchain is composed by three peers (blockchain nodes in Fabric's jargon), one for each federated member which are connected via a Docker Overlay Network [70, 177]. The logical space created upon such infrastructure in Fabric is referred as *channel* (named *sunfish-channel* in our scenario) where *chaincode*s (i.e. smart contracts in Fabric's jargon) are deployed and executed. To support all previously mentioned API, we have developed two main chaincodes:

- `kvs`: offering a key-value store à la Mongo but with decentralisation guarantees;
- `payslip`: offering the decentralised computation of taxes as per UC1 requirement (see Section 4 for further details).

The chaincode functions can be invoked via the `invoke(args)` API. The `kvs` can be easily invoked also via the `put` and `get` APIs.

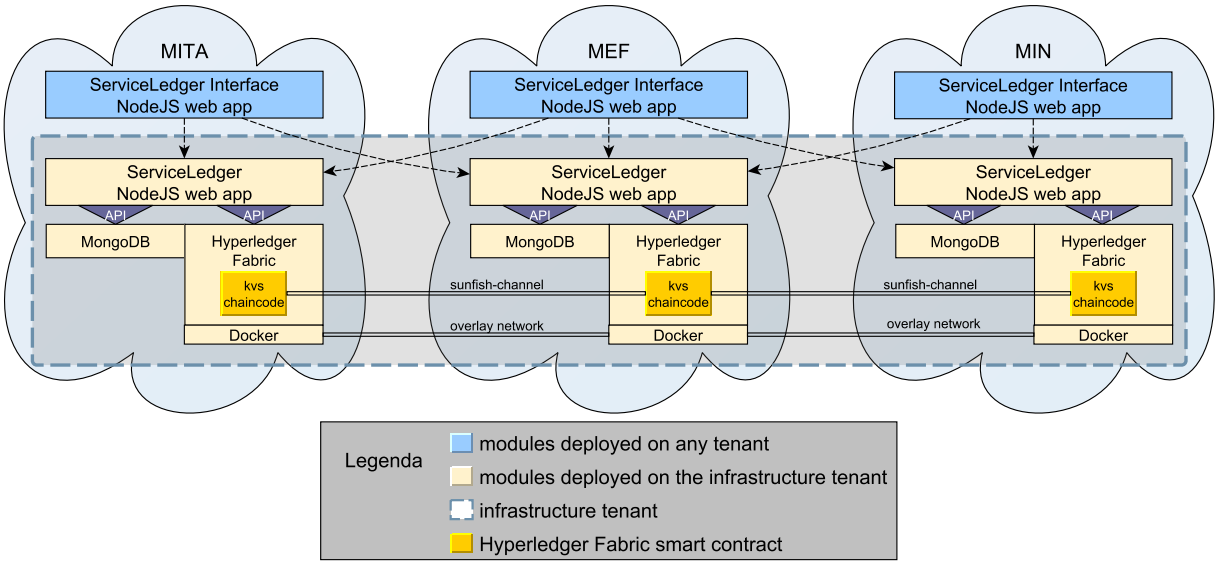


Figure 7.6: *Implementation of the Federated Service Ledger within the SUNFISH federation*

The sources code of the `kvs` chaincode can be found in the SL GitHub repository [49], while we decided to not publish the `payslip` chaincode as it is private and specific for the case study related to MEF and MIN.

7.3 A Smart Contract Solution for a Payslip Computation Use Case in the SUNFISH Federation

In this Section we describe how to employ the Federated Service Ledger infrastructure of SUNFISH for a use case related to the Italian payslip computation involving MEF and MIN. We first describe the use case and its requirements so as to introduce our solution based on smart contract running on top of a permissioned blockchain within the Federated Service Ledger infrastructure.

7.3.1 Payslip Use Case Description and Requirements

The Italian Ministry of Economy and Finance (aka MEF) is currently facing the issue of overcoming segregation of Public Bodies data among Clouds for calculating payslips of Policy Forces. Specifically, the Italian legal framework forces the Ministry of Interior (aka MIN) to be the exclusive controller of Policy Force sensitive data. However, MEF needs to access to such data to correctly compute payslips (for the cognitive, local taxes must be computed on actual residence, which is however sealed for data classification purposes within the MIN).

To overcome this issue, MEF has put in place an intricate cooperation with MIN which locally performs part of the payroll tax computation then to be used by the MEF.

The previous workaround amounted to assume a fake town of residency requirement additional manual effort to realign expected and computed tax at the end of the fiscal year. However, this has led to uncontrolled cooperation prone to mistake and malicious subversions, e.g. to avoid tax payment or to grant huge pay rise all of a sudden. Such frauds are subtle to discover and, most of all, MEF is liable for it even though it has no control on the full payroll data.

Therefore, MEF requires different deployment of such use case to introduce adequate computation guarantees both on the used sensitive data and on the performed computations.

The following use case aims to create a federation between the Cloud systems of MEF and MIN in order to enable the secure processing of such sensitive information for automated calculation of payslip. Indeed, the ultimate goal of deploying a federation-based payroll application is:

"enabling a cooperative calculation of payslips by processing MIN data in an accountable and secure manner, while MEF cannot access the data directly."

Specifically, the MEF's Payroll Application (i.e., the system for payslip calculation of the Italian Public Administration) might be able to exploit data on the MIN database without directly manage such highly sensitive information of the Public Security Department (Polizia di Stato) of the MIN. The data are needed to calculate local taxes on the basis of sensitive information (e.g., home address, work location) that only the MIN can manage so the MEF's Payroll Application applies this rate individually on the salary of each employee making monthly the deduction on payment and providing the money transfer towards the local administrations.

Requirements. We can therefore summarise the business requirements of the applications as follows

1. to ensure correct and timely tax computation of MIN payslip;
2. to ensure use of certified code (provided by MEF) for the computation of tax of MIN payslips;
3. to guarantee that the sensitive data of MIN are disclosed outside MIN's Cloud;
4. to realise accountable computation of tax calculation so to confine MEF liability on the computed data.

Referring to the latest requirement, it is worth noticing that MIN will still be the owner and legal responsible of the correctness of input data for tax calculation (i.e., town of residence, etc.), but MEF must have non-repudiable, accountable means to prove what data MIN used for carrying out specific tax calculation. Therefore, this is clearly confining the liability of MEF on potential maliciously altered payslips.

7.3.2 Smart Contract Solution

Thus, we propose a solution which relies on the Federated Service Ledger of SUNFISH within a permissioned blockchain to allow the payslip computation through smart contract. Specifically, the Payroll application is divided i two parts: the *master* part of the application will run on the MEF tenant, while the *slave* will run on the MIN tenant in the form of a smart contract instantiated and invoked on the local peer of the blockchain.

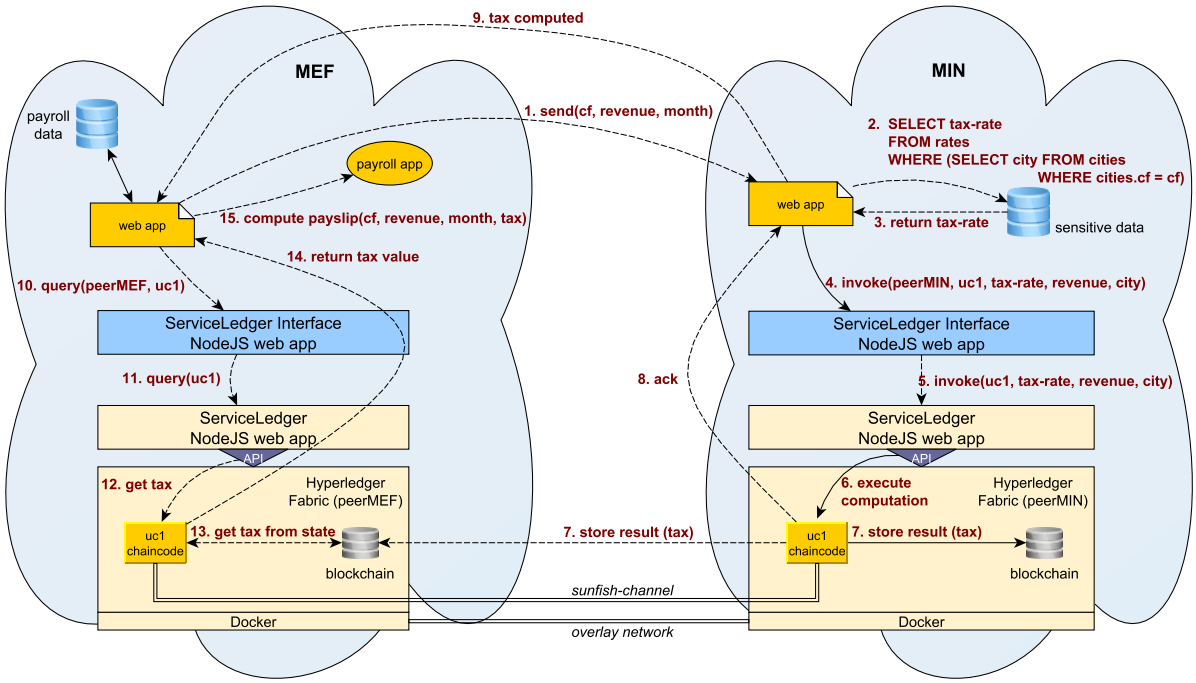


Figure 7.7: *Payslip cross-cloud interactions business flow*

Therefore, each side of the application in MEF and MIN performs different duties:

- Payroll Application (or master):
  - to calculate employee duty time by means of the Fiscal Code;
  - to issue to MIN request of tax calculation;
  - to compile employee final payslip (once MIN tax calculation is received).
- Smart contract (or salve):
  - to select the local home of each employee;
  - to select the rate on basis of town/city code;
  - to calculate the local tax for each employee, which the MEF will get access to via a query to the chaincode.

The Payroll Application has been implemented as a web application deployed on the MEF tenant. The smart contract is instead deployed as a chaincode on top of Hyperledger Fabric. The chaincode is installed only the peer of MEF and MIN and it employs an endorsement policy to make it runnable only by the peer of MIN[3]. The latter endorsement policy is crucial for the correct functioning of the chaincode and the achievement of requirements 3 and 4.

Therefore, we are ensuring that the actual computation of the sensitive data of the MIN does not occur on the MEF side. The MEF, anyway, can query the chaincode to get the calculated tax value, but it will not be able to see the input values. Most of all, the code being part of the blockchain cannot be changed and is indeed certified by both MEF and MIN.

In Figure 7.7 is graphically reported the business flow of the payslip computation for the use case and following we describe each step of the computation.

- Step 1: MEF invokes the computation of the tax salary computation of an employee (identified by its fiscal code cf) for a specific month and along with his gross revenue (it is needed for tax computation).
- Steps 2-3: MIN retrieves all necessary information in its database as taxable income and municipals known by the system.
- Steps 4–6: MIN invokes the smart contract on the blockchain via Service Ledger Interface and Service Ledger. The given arguments are the retrieved information from the database.
- Steps 7–9: the smart contract carries out the computation of tax sending the ack to the MIN (and then to the MEF) and committing the result to the blockchain;
- Steps 10–14: MEF can query the blockchain (via the Smart Ledger) to retrieve the computed tax;
- Steps 15: MEF can now finalise the payslip Consequently, MEF is now also able to provide aggregate data per Municipals in order to make cor-rect payments to local public bodies.

[3]Hyperledger Fabric allows to flexibly tune which set of peer can execute and read values from a chaincode even if they are in the same channel (further information can be found at `http://hyperledger-fabric.readthedocs.io/en/release/endorsement-policies.html`).

Chapter 8

A Blockchain-based Solution to Enable Log-Based Resolution of Disputes in Multi-party Transactions

In the previous chapter we introduced the blockchain as new potential technology to ensure high integrity in a decentralised manner. This feature might play a key role in multi-party transactions. Indeed, as internet-based services are evolving, companies need to integrate their IT infrastructures. Business-to-Business (B2B) integration targets to connect such key business processes in an automated and optimised way, in order to deliver sustainable competitive advantage to customers and suppliers. An example is the electronic purchase of assets, where an enterprise might process in real time order information faster and more accurately, being more responsive to customers, with an improved customer service which may result in sale growth. A relevant example regards, among the others, the aforementioned Cloud federations, where multiple providers share their own resources to cope with load peaks without over-provisioning, by renting out resources otherwise unused.

Such integrations require multi-party transactions that need to be regulated through some SLA so that, in case one party claims a SLA violation, she can prove it. Indeed, each party may keep logs of sent requests and received responses, but the other party may ignore requests/responses or deny logs validity.

Current solutions employ a *trusted-third party* (TTP) [88, 187] which is in charge of checking SLA compliance and solve possible disputes. In this way, parties cannot drop or deny any sent request or received response, because the TTP is involved in and logs every interaction (Figure 8.1). The main drawbacks of TTP-based solutions are mainly related to: (i) performance overhead, as required interactions are routed through the TTP, which can be a single point of failure and a performance bottleneck; (ii) additional fees, as the TTP intermediation does not usually come for free and may ask for an initial fee or for per-transaction fees; (iii) the TTP must be trusted and if it behaves dishonestly or colludes with the other parties, there is no chance to prove the injustice.

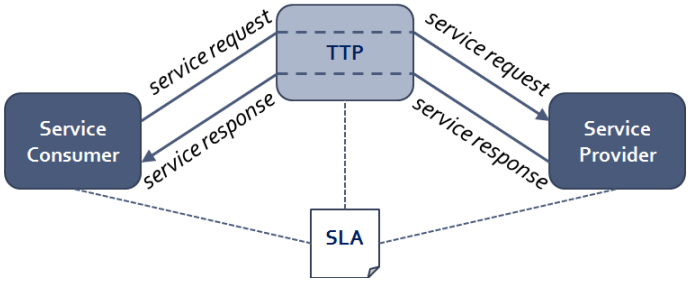


Figure 8.1: *TTP-based solution*

In this chapter we propose *SLAVE* (*Service Level Agreement VErified*), a solution to replace a non-totally trustworthy TTP with an intermediary based on a public blockchain like Bitcoin's [174] or Ethereum [234], such that data sent to a public blockchain cannot be falsified, hence no risk of dishonest behaviour or collusion. Since data in a public blockchain can be seen by everyone, pseudonyms and asymmetric cryptography is used to mask sensitive information.

Contributions. The content of this chapter has been published in [8] and the novelty regards an architecture, and an algorithm in support to it, for securely decentralising the task of a TTP for SLA verification.

Chapter structure. Section 8.1 introduces related works on multi-party dispute resolution. Section 8.2 presents the proposed SLAVE solution, finally Section 8.3 concludes and discusses the work.

8.1 Related Work

SLA is a formal negotiated agreement between a service provider and a customer (see Section 1.3.3) and it is ensured through the service level management (SLM) according to a specified QoS. The monitoring is so delegated to a TTP which monitors the performance status of the offered service. A number of solutions have been proposed to improve monitoring and management of SLAs, their mapping with QoS to ensure and verification [149, 155, 11, 31, 235]. Usually, an user should rely and trust to such systems and there can be further costs to query multiple parties [237]. Often, accountability can result very different and ambiguous from a provider to another even though well-defined SLA [117] making more and more tricky a resolution of disputes which is still based on a trusted party [224, 195, 135]. Conversely to current solutions, our proposal is the employment of a blockchain that in a fully decentralised manner can solve the task without any trust between party.

8.2 SLAVE Solution

In this section we present SLAVE, a solution to enable log-based resolution of disputes in multi-party transactions. SLAVE employs a public blockchain to store requests/responses, thus we

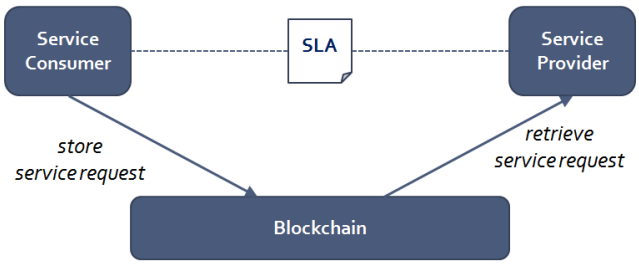


Figure 8.2: *Interaction between a service consumer and a service provider in SLAVE. Requests and responses are stored in the blockchain, they are the logs to be used for dispute resolution*

assume that data stored cannot be tampered with. Both provider and consumer participate in the mining process to detect requests and responses directed to them (see Figure 8.2). Storing requests and responses in a public blockchain provides strong integrity guarantees, thus they can be then used in case of disputes.

As data in a public blockchain can be accessed by everyone, there is the need to mask sensitive information, which in this case are the identities of involved parties and the content of transactions. Identities are so masked through the usage of pseudonyms. Specifically, each party has as many disjoint sets of pseudonyms as the parties it has to interact with, so that each pseudonym is used only to interact with a specific party, which is the only party to know the real identity behind such pseudonym. Each pseudonym is a public key, and the corresponding private key is kept secret by the party itself. We use the notation pk and sk to indicate public and private (i.e., secret) keys, respectively, and the notation $\{m\}_k$ to indicate the encryption of m with a key k.

8.2.1 Phase 1: Handshaking Pseudonyms

For each pair of parties A and B that want to interact through SLAVE, a preliminary handshake phase is required, where A generates a set $\{\langle pk_i^{A,B}, sk_i^{A,B}\rangle\}$ of public/private key pairs to communicate with B, and sends the set $\{pk_i^{A,B}\}$ of generated public keys (i.e., the pseudonyms) to B through a secure channel. Vice versa, B generates a set $\{\langle pk_i^{B,A}, sk_i^{B,A}\rangle\}$ of public/private key pairs to communicate with A, and sends the set $\{pk_i^{B,A}\}$ of generated public keys (i.e., the pseudonyms) to A through a secure channel.

8.2.2 Phase 2: Sending Transaction

Once the handshake phase is completed, A and B can start exchanging transactions using the SLAVE solution. Let T be a transaction from A to B. Let N_T be a nonce computed by A for T to prevent replay attacks. Let $sign(m, sk)$ be the signature computed on (a digest of) message m using the private key sk, used in this case by A to prove the authenticity of its transaction T.

Algorithm 3 SLAVE Send Algorithm

1: **function** SEND(transaction T, private key $ski^{A,B}$, dest B, dest pseudonim list $\{pk_j^{B,A}\}$)
2: $N_T \leftarrow computeNonce(T, B)$
3: $msg \leftarrow \langle T, N_T \rangle$
4: $pk_j^{B,A} \leftarrow getRandomPseudonym(B, \{pk_j^{B,A}\})$
5: $enc \leftarrow encypt(msg, pk_j^{B,A})$
6: $pk_{enc}^{A} \leftarrow encypt(pk_i^{A,B}, pk_j^{B,A})$
7: $sig \leftarrow sign(msg, sk_i^{A,B})$
8: $req \leftarrow createFinalRequest(enc, sig, pk_{enc}^{A}, pk_j^{B,A})$
9: $storeToBlockchain(req)$

The information to be stored in the blockchain also have to include what pseudonyms $pk_i^{A,B}$ and $pk_j^{B,A}$ have been used by A. The former is put in encrypted form, while the latter is kept in clear to let B recognising that the transaction is directed to her and understanding what private key to use to decipher all the data of the transaction.

The Alg. 3 shows the *send* procedure of a party A who wants to send a transaction T to a party B. A adds a random nonce N_T to the original transaction and creates a message $msg = \langle T, N_T \rangle$. Then it choose a pseudonym of B, say $pk_j^{B,A}$. It encrypts then the message msg with the chosen pseudonym, thus $\{msg\}_{pk_j^{B,A}}$. In the same way, It also encrypts its public key with the pseudonym o B. Then, it signs the message msg with its private key to obtain a signature sig. Finally it creates the final request to store to the blockchain as $\langle \{\langle T, N_T \rangle\}_{pk_j^{B,A}}, \quad sign(\langle T, N_T \rangle, sk_i^{A,B}), \quad \{pk_i^{A,B}\}_{pk_i^{B,A}}, \quad pk_i^{B,A} \rangle$.

8.2.3 Phase 3: Receiving Transaction

Algorithm 4 SLAVE Receive Algorithm

1: **upon event** ⟨ new request *req* stored in the blockchain ⟩ **do**
2: $pk_j^{B,A} \leftarrow getPseudonym(req)$
3: **if** $pk_j^{B,A}$ **in** *myPseudonymsList* **then**
4: $sender \leftarrow getSender(pk_j^{B,A})$
5: $pk_i^{A,B} \leftarrow getPublicKey(sender)$
6: $correct \leftarrow verify(sig, pk_i^{A,B})$
7: **if** $correct == 1$ **then**
8: $enc \leftarrow getEncryptedMsg(req)$
9: $msg \leftarrow decrypt(enc, sk_i^{A,B})$
10: $T \leftarrow removeNonce(msg, N_T)$

Every time a new request is stored to the blockchain, providers can detect them (see Alg. 4). Because of $pk_j^{B,A}$, B detects that this request is sent by A to B as only A knows such a pseudonym

of B, hence B first verifies the signature by using A's public key, then decrypt msg with the private key $sk_j^{B,A}$ related to $pk_j^{B,A}$. Finally it removes N_T from msg to obtain the transaction T.

8.3 Discussion

In this chapter we proposed SLAVE, a solution to enable log-based resolution of disputes in multi-party transactions, which replaces the usage of a TTP with a pubic blockchain. SLAVE allows to overcome the limitations of possible malicious behaviours of a TTP, including the risk of collusion with other parties. SLAVE also improves service availability with respect to TTP-based solutions, as thousands of miners supports the blockchain functioning.

As the blockchain provides high latency, the performance bottleneck is still a problem and will be object of study for the next chapter where we first state some open research questions for adopting a blockchain as a distributed database, then we propose an architecture to achieve both performances and high integrity guarantees.

Chapter 9

2LBC: A Tamper-resistant High Performance Blockchain-based Architecture for Federated Cloud Databases

As a critical component of many systems, data have an appealing target for cyber-attacks. Tampering with data can go undetected and drive malicious operations, e.g. data alteration and deletion. Most of all, differently from availability loss, data integrity can be hardly restored once lost. Modern database systems use logging mechanisms to track data changes, e.g. Write-Ahead Logging of PostgreSQL[1] and Redo Log of Oracle[2]. However, if such logging files are forged, recognising an attack or a failure is awkward as data integrity relies too intimately on the system itself. Typically, *Remote Data Auditing* mitigations are employed, but they come with high costs and rely on trusted third parties [214].

Regardless of strong integrity guarantees, exploiting blockchain to strengthen (distributed) database systems implies coping with overwhelming performance penalties: viz., high latency and low throughput. Indeed, preliminary blockchain-based database solutions rationalise blockchain features: BigchainDB [162] replaces PoW (and its security) with a lightweight protocol; RSCoin [64] (re)introduces certain degree of centralisation in the system.

In this chapter we propose 2LBC (Two Layered Blockchain), a blockchain-based architecture for distributed (federated) database transaction (or redo) logs, to achieve both high performance and adequate integrity guarantees. To inhibit possible collusion attacks and to avoid blind trust on the integrity guarantees claimed by cloud providers, we advocate an innovative exploitation of the blockchain. Specifically, we define a layered architecture that makes use of a *first layer block-*

[1] `postgresql.org/docs/9.1/static/wal-intro.html`

[2] `docs.oracle.com/cd/B19306_01/server.102/b14231/onlineredo.htm`

chain assuring low latency and high throughput, anchored to a *second layer blockchain* featuring PoW to ensure data integrity. As a case study, we address the distributed database underlying the SUNFISH FaaS [205]. Through an experimental evaluation on a small private datacenter, we assess the effectiveness of a real prototype implementation, which features a total consensus algorithm on the first layer blockchain and Ethereum as a second layer.

Contributions. The content of this chapter has been published in [89, 7]. The novelty of the proposed solution can be summarised as follow:

- we stand some open research questions regarding the blockchain technology and we propose some answers to cope with them;
- we propose 2LBC, an architecture based on two layer of different blockchain to achieve both performance and integrity guarantees in a cloud federation;
- we propose an implementation and experimental evaluation of a real prototype;
- we propose a solution based on a Distributed Hash Table to shard the ledger among the network instead of replicating the ledger.

Chapter Structure. In Section 9.1 we introduce the state of the art, by summing up some open research questions and proposing related answers. In Section 9.2 we introduce the system and threat model and in Section 9.3 we present the 2LBC architecture. Sections 9.4 and 9.5 show instead respectively the prototype implementation and the experimental evaluation. Finally, Section 9.6 mark out a solution to improve scalability through a Distributed Hast Table solution and Section 9.7 concludes and discusses the work.

9.1 State of the Art

Data Integrity is a well known problem in computing systems, especially for Cloud environments, where users outsource their data. The task of checking data integrity for a user having relatively poor computing devices might be very heavy due to huge amount of data to download. Ateniese et al. [15] provide one of the first model that enables a client to verify the integrity of her outsourced data on a single server without retrieving them. For a Cloud environments, Remote Data Auditing (RDA) is a solution to enable auditability of outsourced data through a trusted third-party, which alleviates the computation burden on the user. A number of RDA techniques have been proposed to improve both security and efficiency, as Sookhak et al. reviewed and classified in their survey [214].

9.1.1 Related Work on Blockchain

All aforementioned works to ensure data integrity rely on the assumption that the third-party is trusted. If the latter acts instead maliciously, they can no longer ensure integrity. Blockchain

allows to decentralised the auditability of data, however in the state of the art a number of works have been proposed to improve blockchain performances. The first-layer blockchain we use to improve performance is inspired by Bitcoin-NG [81], a modified version of the Bitcoin protocol which is scalable. They specifically, to improve the performances employ two kind of blocks and for separated tasks: *key blocks* for leader election and *microblocks* to collect transactions. They modified the trust of the network and sacrifice some security guarantees, indeed data integrity is ensured only under the assumption of a majority of honest miners.

GHOST is another solution to improve the scalability, proposed by Sompolinsky et al. [213]. GHOST forces more transactions to the network by edit the rule for accepting the main valid blockchain. Every time a fork occurs, GHOST chooses the branch having the most amount of work as the valid chain. Thus, GHOST not only accepts the earliest block at each epoch, but also other blocks found later, i.e. "*orphaned blocks*". GHOST allows block parallelism in the network.

An improved consensus protocol which scale better than Bitcoin-NG and GHOST is SCP [156]. They randomly place nodes in several committees which run a classical consensus protocol within each committee to propose blocks in parallel and they show as the throughput scales quite linearly with the computation.

Other blockchain-based databases have been proposed in literature. A similar work to ours is BigchainDB [162], a NoSQL-like storage on top of a blockchain with a built-in consensus approach. Similarly to BitCoin-NG, their main goal is to improve performances sacrificing some security guarantees. Indeed, in case of a majority of malicious miners, they can no longer ensure data integrity, similarly to Bitcoin-NG. Our solution, contrarily from both Bitcoin-NG and BigchainDB, can ensure data integrity even in case of a majority of malicious miners. Indeed, by taking advantage of the PoW-based blockchain immutability feature on the second layer, our solution is able to ensure data integrity also when all the miners but one are malicious and collude among themselves.

Similarly to the anchoring we employed, Sidechains is a protocol which permits users to move coins between different blockchains [24]. It is well clear as Sidechains neither solves scalability problem in Bitcoin nor reduces any costs, but is just but a solution to easily setting up experimental blockchain without requiring a currency. Furthermore it does not improve intergrity guarantees.

A remarkable work aimed at improving performance while providing security guarantees is RSCoin [64], i.e. a cryptocurrency framework that introduces a centralisation degree. A central bank maintains complete control over the monetary supply, but relies on *mintettes*, i.e. a distributed set of authorities used to prevent double-spending. They show how their solution, based on a proper consensus algorithm, allows to improve performances and ensure integrity. However, compared to our solution, their work has two main limitations: (i) a centralisation degree and (ii) integrity guarantees reached only with a majority of honest mintettes.

9.1.2 Open Research Questions

In the context of cloud computing environments, the blockchain could be exploited to realise a database ensuring strong integrity guarantees. In particular, the blockchain could be used to store the logs of database operations, thus to avoid threats to data integrity. However, current blockchains cannot be employed "as-is" due to various deficiencies. In the following, we address the main issues related to data integrity, performance limitations, and blockchain stability.

How to Measure Data Integrity? Once data has been included in a block, if we assume a majority of honest miners, we can be highly confident that the chances of data alteration decrease exponentially over time (see Sec. 7.1). Data integrity is indeed strictly related to these chances. The more unlikely it is that data can be tampered with, the stronger the integrity guarantees we can claim. However, being dependent on time introduces critical aspects to address for ensuring data integrity: there is hardly information on the effective integrity guarantees once a transaction has just been sent to the blockchain. These observations suggest that data integrity on blockchain cannot be simply seen as a binary property, which either holds or does not, but it should be intended as a more complex, quantitative concept. This amounts to take multiple factors into account, including the time and parties' awareness. The described issues thus lead us to formulate this research question:

Q1 *How can we quantitatively characterise data integrity guarantees, in order to enable comparison among different blockchain-based database solutions?*

Reasonable approaches to answer to this question should be based on the the effort an attacker would spend to compromise data integrity without being detected.

How to Improve Performance? The performances currently achievable with PoW-based blockchains are really poor as compared with classical database technologies. The experimented latency and throughput are almost incompatible with the requirements of the considered cloud scenarios. In this sense, a challenging and fundamental research problem regards the investigation of novel blockchain designs aimed at delivering performances aligned to today's requirements, while keeping the needed integrity guarantees.

Q2 *How can we design a blockchain-based database with better performances compared to a PoW-based blockchain "as-is", and with comparable data integrity guarantees?*

The resulting designs should be flexible enough to enable variable tradeoff between performance and integrity, thus to choose the setting that better fits the requirements.

How to Enhance Stability? Current PoW-based permissionless blockchains rely on a market-dependent cryptocurrency that may make the storing of data highly expensive and too dependent on market variations, i.e. it cannot ensure stability. Likewise poor performance cut out many

possible practical applications, unsatisfactory stability assurances can severely restrict the applicability of blockchain to database. Enhancing the stability of PoW-based permissionless blockchains, e.g. Bitcoin's, amounts to alter the currency incentives underlying the mining process. Due to the large economic interests and speculation behind cryptocurrencies, such an amendment is practically infeasible. A more viable path is exploiting permissioned blockchains, where incentives do not depend on cryptocurrencies. The described path corresponds to answering the following research question:

Q3 *How can we setup a permissioned blockchain having stronger stability compared to existing PoW-based blockchains, while preserving required guarantees on data integrity?*

The resulting blockchain will guarantee a stable support for the development of distributed databased, as it is needed, e.g., in cloud computing environments.

9.1.3 Preliminary Answers to the Research Questions

In the following, by referring to our proposal for blockchain-based databases, we introduce our preliminary answers to the research questions previously reported.

Measuring Data Integrity. Our answer to Question **Q1** is to measure the integrity as the effort required for an attacker to change data in the blockchain without being noticed. Quantifying this measure highly depends on the nature of the considered blockchain, but in general a desiderata is that the longer data are stored in a blockchain, the greater the effort an attacker should pay to break data integrity.

In our two-layer blockchain, the integrity measure on the evidences of a database operation is crucially affected by which of the chain contains the evidences. Indeed, if the evidences are only on the first-layer, the effort required for an attacker corresponds to compromise all the replicas of the first-layer. However, as soon as the hash of the corresponding evidences has been stored in the second-layer, an attacker should also subvert the integrity of the PoW, thus the data integrity measure would be higher. As a matter of fact, on PoW-based blockchains like Bitcoin's, the attacker effort is close to infinite [90], i.e. it is an infeasible attack.

Improving Performance. The lack of performance of current blockchain-based systems, i.e. Question **Q2**, is due to PoW. To provide better performance to blockchain users, our proposal offers to clients a blockchain based on lightweight consensus algorithm and leverages on the power of PoW only in the background, i.e. the second-layer. Therefore, from the point of view of a client of the blockchain-based database, an operation on the database is completed as soon as it is elaborated by the first-layer blockchain.

Enhancing Stability. Permissionless blockchains like Bitcoin's and Ethereum's are natural candidates for the second-layer blockchain. To enhance their stability, i.e. Question **Q3**, we advocate

the use of a blockchain that does not feature a market-dependent cryptocurrency and mining incentive mechanisms. Broadly speaking, the stability needs for blockchain highly depend on the application context. For example, in the case of the SUNFISH case studies, which address the European public sector, the need of stability is of paramount importance. In this context, we could hence envision a European permissioned PoW-based blockchain that will offer a common, stable underlying support for all European administrations.

9.2 System and Threat Model

9.2.1 System Model

We consider a cloud federation composed by M members. Each federation member employs a node to set up a distributed system with M nodes connected by a network. We refer to these nodes as *miners*. Miners can be heterogeneous, so with different hardware, computational power, memory and bandwidth. Messages from correct miners cannot be lost by every correct miner because they can communicate each other by using a point-to-point reliable link or a reliable broadcast primitive. The system is synchronous, hence miners share the same global clock and messages from correct miners are certainly delivered after a maximum delay Δ. We consider the continuous time to be divided into blocks. We refer to such blocks as *round* and an increasing number identifies each round. Being the system synchronous, we assume miners to share the same round. The change of a round is triggered by a specific event (e.g. a timeout) and involve every miners simultaneously.

We assume a Byzantine failure model, in which an attacker may subverted up to f miners, that so may misbehave arbitrarily. The network is static as faulty miners remain in the network and new miners cannot join the network.

We consider the distributed system providing a fully-replicated key-value store database service through a *State Machine Replication* model, thus update operations are driven by a consensus algorithm among miners.

A set of *clients* can interact with the database by sending *request operations*. We consider that request operations lead *transactions* toward the database. The complete transaction life-cycle is shown in Figure 9.1 and consists in the following states:

- *pending*, when transaction has been created and miners put it on the queue;
- *refused*, when miners refuses the transaction considering it malicious;
- *verified*, when miners accepted the transactions considering it as correct;
- *completed*, when the transaction has been verified or refused and has been added in the first-layer blockchain.

A further *finalized* state can be considered when a transaction has been witnessed on a permissionless blockchain[3]

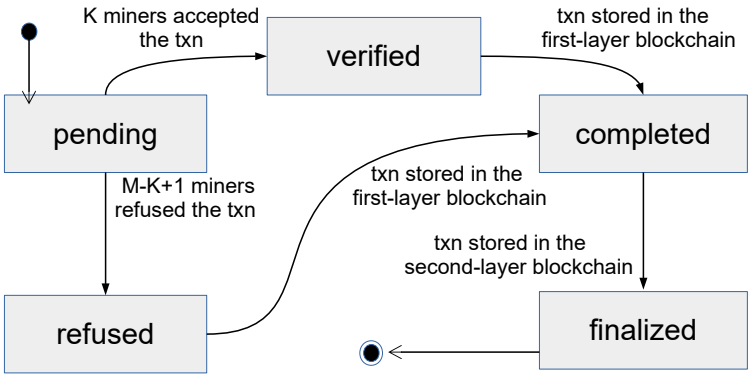


Figure 9.1: *Transactions State Diagram*

Miners and clients have a couple of public-private keys respectively pk_mi and sk_mi (pk_c and sk_c for the client) and everyone knows other public keys.

The problem we tackle in this work is the definition of a replicated database with consistent data between nodes in presence of a number of Byzantine clients and/or miners. Specifically, the main goal is to provide strong data integrity even under a collusion attack, i.e. when all miners have been compromised. Possible malicious behaviours of the attacker are detailed in next subsection.

9.2.2 Threat Model

We consider a very strong adversary able to coordinate other faulty nodes, delay messages and falsify messages to cause the most damage to the system. We assume that the adversary and the faulty nodes are computationally bound so, with high probability, they cannot break the cryptographic techniques mentioned above. Therefore, the adversary cannot create a valid signature of non-faulty nodes unless she steals their private keys.

We consider that the public permissionless blockchain, being based on proof-of-work, is able to produce an impractical mining hash power for an attacker, hence we assume that data in the public blockchain cannot be broken.

In the context of the considered SUNFISH case study, the focus is on the database storing the governance data of a federation, hence on data whose corruption critically affects the whole federation and its security. The threats we consider range from attacks to data integrity as malicious alterations of a database datum or updating of data without informing all the involved federation members, to *denial-of-service* (DoS) attacks. Specifically we modelled three possible kinds of attack:

[3]Note: this state is required from our solution; this will be detailed in Section 9.3.

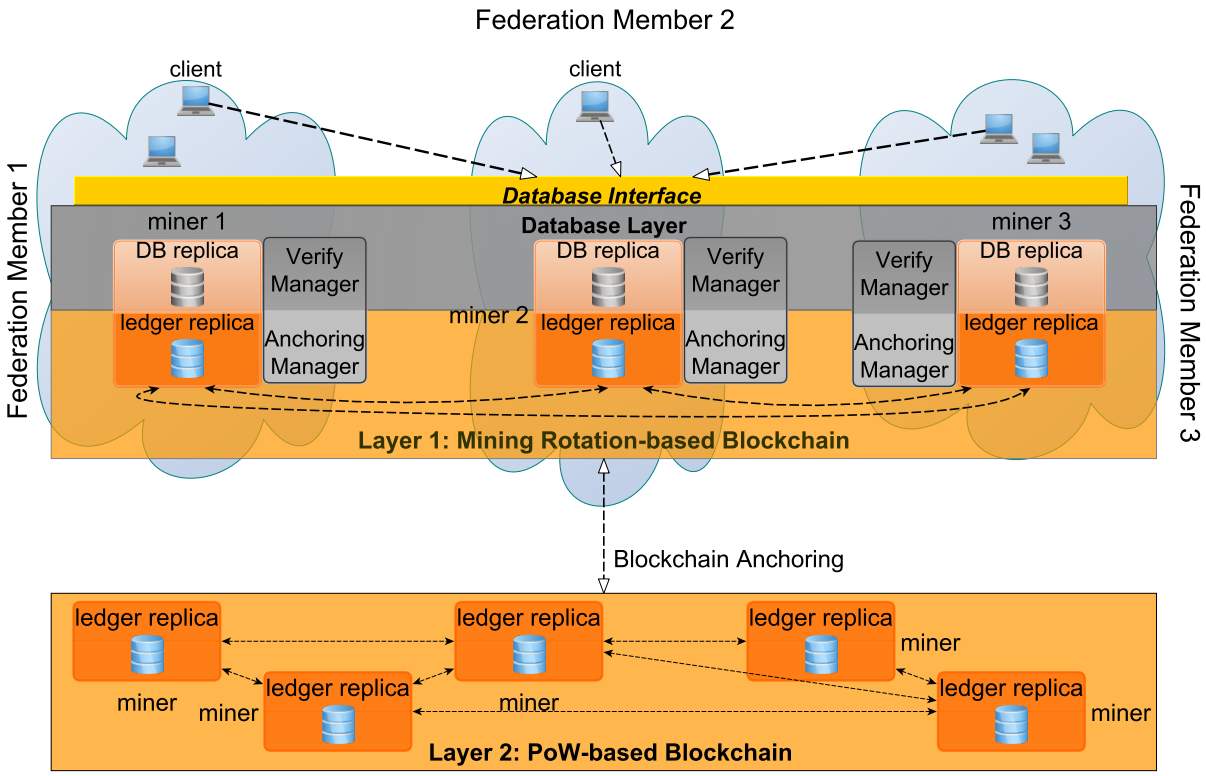


Figure 9.2: *A blockchain-based database proposal for a Cloud Federation*

A1 (*tampering*): if a miner tries to modify the log compromising the integrity of it. Specifically there can be three possible threats:

- **T1** An attacker violates the integrity of the data by directly altering (part of) the database.
- **T2** A federation member updates the database without informing the other members.
- **T3** Multiple federation members collude to maliciously altering (part of) the database.

A2 (*forgering*): if a miner tries to send a fake txn, i.e. by forging a client operation;

A3 (*DoS*): if a miner either does not sign a txn or a leader does not broadcast a txn.

9.3 2LBC Architecture

In this section we introduce our proposal for an effective blockchain-based database. Blockchain is appropriately exploited to ensure the integrity of distributed replicas of a database, i.e. to store persistent evidences of the database operations that cannot be repudiated. The use of blockchain ensures not only integrity guarantees, but also fully distributed control of the database data. This intrinsic characteristic makes our proposed database feasible to be used in the context of FaaS federations. Figure 9.2 graphically depicts the proposed blockchain-based database distributed on three clouds member of a federation.

Clients of the blockchain-based database issue operations through the *Database Interface*. The operations are first logged via appropriate evidences by the the first-layer blockchain, then they are executed on the distributed *DB replica*s. More specifically, the first-layer blockchain is permissioned, and features one miner on each member cloud. The miners, by relying on a public/private key pair to sign messages, achieve consensus by means of the so-called *mining rotation* consensus mechanism. Namely, it divides the time into rounds and, for each round, elects a miner as a leader. The leader is then in charge of receiving new operations, signing them with its private key, and broadcasting them to the other miners. Once all miners have signed the operations, they can become part of the blockchain: all the miners add these operations to their local *ledger*, and apply them to their local replica.

The interaction with the second-layer PoW-based blockchain is realised via a *blockchain anchoring* technique. The anchoring technique is a timed operation that permits linking a specific (part of) the first-layer blockchain with (a block of) the second-layer blockchain. In particular, at certain intervals of time, a *witness transaction* containing the hash of the first-layer blockchain up to the current operation is sent to the second-layer blockchain and, consequently, stored as immutable, irreversible transaction. These hashes act as forensics evidence for proving and validating the integrity of the data stored in the first-layer blockchain.

9.4 Prototype Implementation

We developed the first layer blockchain in Java. Miners join a p2p network and communicate through txns composed by a *payload* (i.e., client *operation*, a *sequence number* and a *timestamp*) and a *certificate* with miners' signatures. Miners trade messages either directly through JavaRMI or in broadcast through JGroup[4]. Clients can operate on the database by issuing to all miners, through JavaRMI, two kind of *operations*: (i) `set(k,v)` to assign a value `v` to a key `k` (it returns to the client a boolean confirmation); (ii) `get(k)` to obtain the value stored to key `k` by returning the txn related to the last `set` toward `k`.

9.4.1 Consensus

In case of `set(k,v)`, miners have to achieve consensus to give confirmation to the client. The consensus is based on a *three-phase commit* protocol [211]. Practically, a client broadcasts a $\texttt{set}_i$ operation to all miners which verify its correctness through a *Verify Manager* and add it to a queue. Once the current leader (who is in charge of ordering txns) proposes $\texttt{txn}_i$ related to $\texttt{set}_i$, other miners remove it from their queue and broadcast their signatures to build the corresponding certificate. When the certificate contains all the signatures, all miners *commit* the txn in their ledger replica and trigger the update in the database replica.

In case of `get(k)`, every miner answers to the client with the last txn related to a `set(k,v)`.

[4] `http://jgroups.org/`

The client can obtain the value v from the most recent txn among those received, and verify its correctness via the miners' signatures.

9.4.2 Anchoring

When the round time of a leader terminates, it triggers a *leader change*, which amounts to store a special txn, and the anchoring procedure. Specifically, the abdicating leader computes the SHA-1 hash of the first layer blockchain and sends a *witness txn* to a dedicated *witness smart-contract* on the second layer implemented with Ethereum.

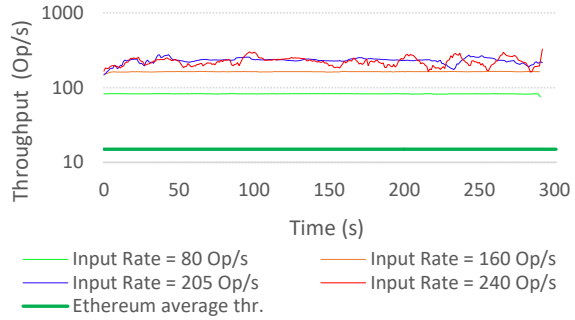
9.5 Evaluation

9.5.1 Security Analysis

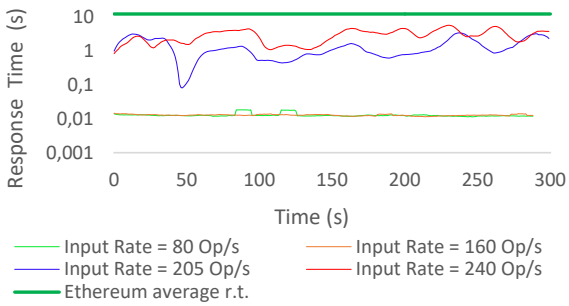
In our approach a valid txn must contain a certificate with all miners' signatures. This avoids miners from maliciously update values without informing other members (**A1**), and from creating fake txns (**A2**) (unless it can obtain the private keys of all other miners). Moreover, hashes stored in the second layer blockchain are immutable, hence, though the attacker is able to compromise the first layer by stealing all miners' keys, **A2** might be detected by comparing the hash of the first layer with the hash evidence stored on the second layer; to compromise also hashes in the second layer the attacker should obtain a significant, unfeasible computational power [90].

Specifically, Threat **T1** is straightforward, while Threat **T2** is due to the democratic nature of a SUNFISH federation. For instance, adding to the database a log entry about a fake inter-cloud interaction between members A and B, hence without the A and B being informed, is a clear integrity violation. Therefore, the process of adding data to the database should rely on opportunely devised consensus schemas. However, as reported in Threat **T3**, even consensus schemas can be attacked: federation members can collude together to alter the database integrity. For instance, given a member A providing to the federation a service s, the other members can collude to compromise s by, e.g., storing false information on the service (i.e., altering the contracts regulating the provisioning of s, or removing log entries about inappropriate uses of s) to obtain advantages and causing the detriment of A. Anyway, the leader cannot forward fake txns (**A2**), because miners sign only txns related to operations that they enqueued. Specifically, miners verify the correctness of txn fields being sure that are not forged.

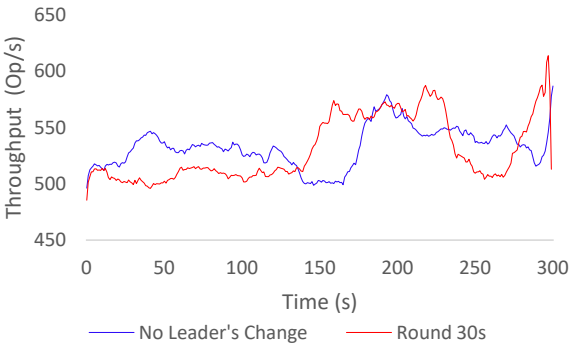
This solution is instead vulnerable to DoS attacks (**A3**): a single malicious miner can block `set` operations not sending its signature or a malicious leader (during its leading periods) can avoid to forward a txn. However, DoS cannot happen with `get` operations, as miners return their response independently. Indeed, if there is at least one honest miner, a client can obtain the value by its corresponding txn and verify the signatures in the certificate to prove the authenticity.



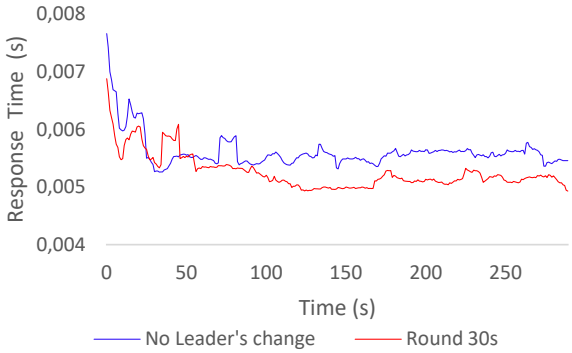
(a) Throughput of `set` operations for different input rates



(b) Response Time of `set` operations for different input rates



(c) Throughput for `get` operations while injecting ≈ 500 op/s without changing leader and with 30s leader round time



(d) Response Time for `get` operations while injecting ≈ 500 op/s without changing leader and with 30s leader round time

Figure 9.3: *Throughput and Response Time evaluation of 2LBC for* `set` *and* `get` *operations*

9.5.2 Experimental Evaluation

We evaluated our prototype on a private cluster composed by 6 Ubuntu 16.04 Virtual Machines (VMs) each one running a miner process, deployed on 4 blade servers IBM HS22, equipped with 2 Quad-Core Intel Xeon X5560 2.28 GHz CPUs and 24 GB RAM.

Network latency between miners is simulated according to a Poisson distribution in the range 5-20 ms. Operations `set` and `get` on random keys are injected with different rates via 2 multi-thread clients deployed on 2 further VMs. To evaluate 2LBC performances, we measured throughput and response time of the operations over time.

For `set` operations, Figure 9.3(a) (resp., 9.3(b)) reports throughput (resp., latency) results that, as expected, are much higher (resp., lower) than Ethereum. Above 240 Op/s, they become unstable for resource saturation due to exchanged messages: the total consensus algorithm requires miners to trade all their signatures for each operation. Additionally, leader rotation introduces an overhead on the enqueued operations during leader changes; we refer to *transient state* as the transitory period where latency exceeds 50% of the average latency. As shown in Figure 9.4, increasing the round time makes the transient state shorter. On the contrary, frequent leader changes increase such overhead.

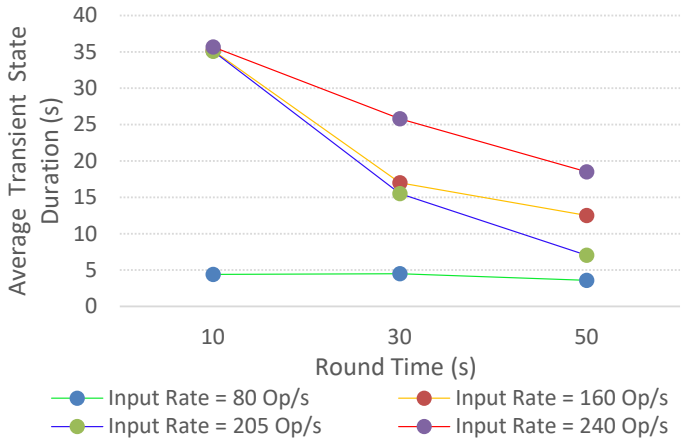


Figure 9.4: *Analysis of the relation between round time (x axis) and transient state duration (y axis) for different input rates*

For `get` operations, Figures 9.3(c) and 9.3(d) report throughput and latency results. As there is no transient state for `get` (i.e., miners just send their last local value), the results are comparable with and without leader rotation.

9.6 Improving Availability and Scalability

The main limitations of the current 2LBC prototype are related to *availability* issues (as explained in last section) and *scalability*. Indeed, overall system performance does not scale adding new nodes, as the used total consensus algorithm has lower performance with additional nodes. In this section we propose the solutions to cope with them.

Availability. To mitigate such limitations, a Byzantine Fault Tolerant solution based on PBFT [47] can be employed. To tolerate up to f Byzantine miners it requires $3f + 1$ miners to provide both *safety* and *liveness*. Note that, however, for the FLP impossibility [84], during periods of asynchrony the protocol may stall. Otherwise, compared to proposed total consensus based on 3PC, PBFT leads to higher availability level as honest miners need to wait just $f + 1$ valid signatures to commit a txn. We might so tolerate up to f *silent* miners at the cost of weaker integrity guarantees; indeed data corruption is possible with just $f + 1$ compromised miners, rather than all N when total consensus is used. Anyway, the integrity of txns already witnessed in the second layer is still ensured thanks to the PoW.

Scalability. To improve scalability, we propose a *data sharding* solution for permissioned (federated) blockchain, similar to [157] for permissionless blockchain. Specifically, we introduce a DHT-based ledger in which each miner, on the base of a *keyspace partitioning*, only handles txns for specific subsets of keys. This approach permits tuning txn loads on miners and, consequently, makes the system more scalable. Furthermore, each key range has a configurable replication factor to en-

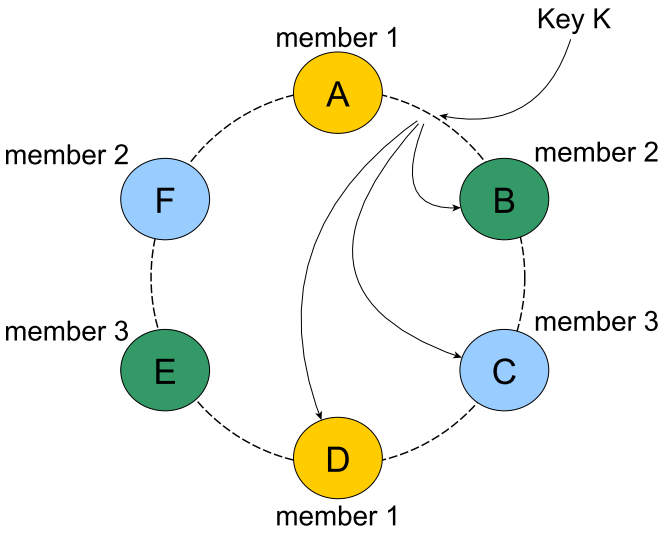


Figure 9.5: *Example of the DHT-based ledger solution to achieve total replication in the FaaS scenario. The federation is composed by $M = 3$ members (each one marked with a proper color), each one exposing $N = 2$ miners (identified by a proper letter). Miners are disposed on the ring so as each member has all keys of the database sharded between its miners. The single miners keep only a subset proportional to the replication factor. In the example, the replication factor is $M = 3$ and each miner maintains only $1/N = 1/2$ of database keys.*

able fault tolerance. In our solution, contrarily to common DHT implementations, the miners involved in a `set` operation must achieve consensus before writing the operation in the local replica (hence in the local keyspace) of the redo log. This permits achieving strong consistency, hence avoiding consistency issues which mar well known DHT-based NoSQL databases, such as DynamoDB (`aws.amazon.com/dynamodb/`) and Cassandra (`wiki.apache.org/cassandra/`). Clearly, this comes at the cost of performance penalties, whose quantification depends on the technology and is beyond our scope here.

The solution to the FaaS scenario can e.g. change as follow: let M be the number of federation members, each providing N (rather than a single) miners. The DHT ring includes $M \cdot N$ nodes, the replication factor is set to M, and miners can be placed over the ring so that the miners of each federation member collectively manage all the keys. Figure 9.5 shows an example of a FaaS federation composed by $M = 3$ members, each exposing $N = 2$ miners.

9.7 Discussion

In this chapter we stated the main open research questions with the related proposed answers for adopting a blockchain as a distributed database. In order to achieve both high performances and integrity guarantees we proposed 2LBC, a two layered architecture which relies on a permissioned

and a permissionless blockchain. An experimental evaluation backup showed the effectiveness of the proposed solution. The main limitations of our approach are mostly related to *availability* issues and *scalability*, indeed, adding nodes does not improve the overall system performance, but instead degrade them, due to the nature of the used consensus algorithm. For the scalability we propose a DHT-based solution to shard the ledger among nodes. To mitigate availability, we propose to replace the total consensus algorithm with PBFT-based.

In next chapter we analyse BFT consensus alternatives to those considered. Specifically, to better fit the SUNFISH requirements of decentralisation, we assess a new family of arising consensus algorithms which make still use of leader rotation, namely Proof-of-Authority family.

Chapter 10

Assessing Proof-of-Authority Consensus for Permissioned Blockchain through a CAP Theorem-based Analysis

Recently, a new line of research on *distributed consensus* is arising, as it plays a key role in the blockhain ecosystem. In permissioned settings, dominant candidates are *Byzantine fault tolerant* (BFT) algorithms such as the Practical BFT (PBFT) [47]. Indeed, BFT-like algorithms have been widely investigated for permissioned blockchains [222] with the aim of outperforming PoW while ensuring adequate fault tolerance.

Proof-of-Authority (PoA) [182] is a new family of BFT algorithms which has recently drawn attention due to the offered performance and toleration to faults [7, 89, 100]. Intuitively, the algorithms operate in rounds during which an elected party acts as *mining leader*, being in charge of proposing new blocks on which distributed consensus is achieved. Differently from PBFT, PoA relies on significantly less message exchanges hence offering significant better performance [69]. However, the actual consequences of such message reduction is quite blurry, especially in terms of offered availability and consistency guarantees with a realistic *eventually synchronous* network model (the Internet is deemed such [41]).

To this aim, we take into account two of the main PoA implementations, named Aura [16] and Clique [53], which are used by Ethereum clients for permissioned-oriented deployments. In particular, the lack of appropriate documentation and analysis prevents from a cautious choice of PoA implementations with respect to provided guarantees, fault tolerance and network models.

In this chapter, we first derive the actual functioning of the two PoA algorithms, both from the scarse documentation and directly from the source code. Then, we conduct a comparison with PBFT in terms of security and performances. Specifically, we conduct a qualitative analysis in terms of the CAP theorem [96], i.e., by assessing which "*two out of three*" properties between *consistency*, *availability* and *partition tolerance* can be assured at the same time. Finally, we assess

performances in terms of message exchange. The analysis assumes an *eventually synchronous network* and the presence of Byzantine nodes. The conducted analysis results that PoA algorithms favour availability over consistency, oppositely to what PBFT guarantees. In terms of latency, measured as the number of message rounds required to commit a block, PBFT lies in between Aura and Clique, outperforming the former and being worse than the latter. These results suggest that PoA algorithms are not actually suitable for permissioned blockchains deployed over the Internet, because they do not ensure consistency, and strong data integrity guarantees are usually the reason why blockchain-based solutions are employed. We advocate that PBFT is a better choice in this case, although its performance can be worse than some PoA implementations, thus their application may highly depend from the scenario.

Contributions. The content of this chapter has been published in [215]. The contributions can be summarised as follow:

- we detailed two of the most prominence version of PoA algorithms by reversing source code and online documentation;
- we proposed an approach to evaluate consensus algorithms through the CAP;
- we conducted such an analysis on the two PoA algorithms as well as on PBFT;
- we analysed performances of PoA and PBFT.

Chapter Structure. Section 10.1 introduces background concepts and comments the closest related work. Section 10.2 introduces the PoA consensus schema. Section 10.3 analyses PoA algorithms with respect to ensured guarantees and performance. Finally, Section 10.4 concludes the work.

10.1 Related Work

Consensus is a well-known problem of distributed computing. It consists in achieving an agreement among a distributed number of processes [41]. Among others, a prominent consensus schema is so-called Byzantine Fault Tolerant (BFT). Protocols of such type are able to tolerate arbitrarily subverted nodes trying to hinder the achievement of an consistent agreement.

The *Practical Byzantine Fault Tolerance* (PBFT) [47] is one of the most well-established BFT algorithms. Specifically, it rests on three rounds of message exchange before reaching agreement. This ensures that $3f + 1$ nodes can achieve consensus also in presence of f Byzantine nodes; this is proved to be optimal [47]. Many other BFT algorithms have been proposed, mainly for improving PBFT performance; among others we can cite *Q/U* [2], *HQ* [62], *Zyzzyva* [138], *Aardvark* [52], see the survey in [222] for further details.

The wide interest on blockchain has prompted substantial research efforts on distributed consensus schema, comprising also new ad-hoc BFT ones. In [160] the author reviews well-known families of consensus algorithms for both permissionless and permissioned blockchains. This

includes Proof-of-Work (PoW), Prof-of-Stake (PoS), Delegated Proof-of-Stake (DPoS), Proof-of-Activity (PoW/PoS-hybrid), Proof-of-Burn (PoB), Proof-of-Validation (PoV), Proof-of-Capacity (PoC or Proof-of-Storage), Proof-of-Importance (PoI), Proof-of-Existence (PoE), Proof-of Elapsed Time (PoET), Ripple Consensus Protocol and Stellar Consensus Protocol (SCP). Although each algorithm is briefly described, they lack of any form of analysis in presence of Byzantine nodes under an eventual synchronous model.

Understanding the most appropriate consensus algorithm among the plethora before is a challenging task that a few works have tried to tackle. For instance, Sankar et al. investigates in [204] the main differences between SCP and consensus algorithms employed in R3 Corda and Hyperledger Fabric. Similarly, Mingxiao et al. extensively compare in [167] performances and security of PoW, PoS, DPoS, PBFT and Raft. While, Tuan et al. propose in [69] a practical benchmark for blockchain, named Blockbench, to systematically compare performances, scalability and security of multiple blockchain systems.

From a more formal perspective, Vukolić compares in [229] PoW with BFT-like approaches introducing the distinguishing property of *consensus finality*: the impossibility of reaching consensus without fully distributed agreement. In blockchain's jargon, it amounts to the impossibility of having forks. As expected, PoW does not enjoy consensus finality (as forks can happen), while all BFT-like approaches does (all parties reach agreement before consensus).

More related to permissioned blockchain, Cachin and Vukolić propose in [42] a thorough analysis of most-known permissioned systems and their underlying consensus algorithms in term of safety and liveness guarantees under eventual synchrony assumption. This work firstly introduces a structured comparison among consensus algorithms, but it overlooks consistency and availability guarantees ensured by their usage; most of all it does not address PoA.

To sum up, none of the aforementioned works discuss the PoA consensus family. To the best of our knowledge, we believe this is the first work tackling the analysis of blockchain consensus algorithms from the perspective of the CAP theorem.

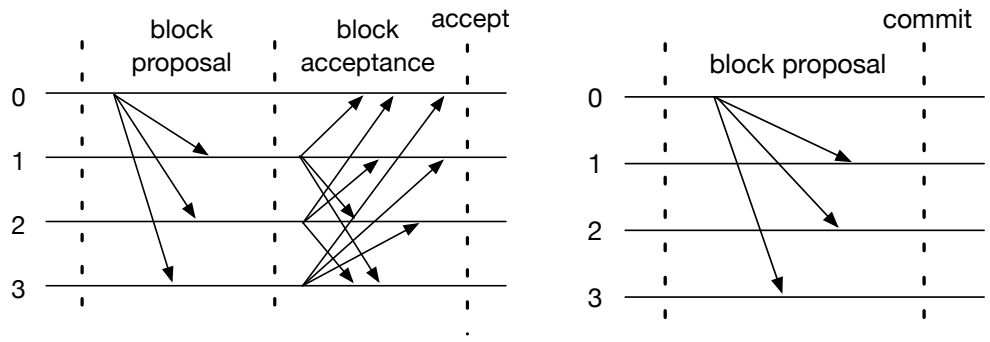
10.2 Proof-of-Authority Consensus

Proof of Authority (PoA) [182] is a family of consensus algorithms for permissioned blockchain whose prominence is due to possibly increased performances than typical BFT-based, resulting from lighter message exchange. PoA has been originally proposed by a team of Ethereum for private networks within two implementations of its client, namely *Aura* and *Clique*.

PoA algorithms are run by a set of N trusted nodes, i.e. the *authorities*. Each authority is identified by a unique *id*. A majority of honest authorities is assumed, i.e. at least $N/2 + 1$ [1]. The authorities run a consensus to order the transactions issued by *clients*. Consensus in PoA algorithms relies on a *mining rotation* schema, a widely used approach to fairly distribute the responsibility of block creation among authorities [100, 89, 7]. Time is divided in *steps*, each one having an author-

[1] The symbol '/' indicates the Euclidean division, which has the quotient of the operands as result.

ity elected as the leader in charge of imposing transaction ordering, i.e. proposing a new block to the other authorities. The two PoA implementations work quite differently: both have a first round where the new block is proposed by the current leader (*block proposal*); then Aura requires a further round (*block acceptance*), while Clique does not. Figure 10.1 depicts the message patterns of Aura and Clique, which will be detailed in next subsections.



(a) Aura (b) Clique

Figure 10.1: *Message exchanges of Aura and Clique PoA for each step. In this example there are 4 authorities with id 0,1,2,3. The leader of the step if the authority 0.*

10.2.1 Aura

Aura (Authority Round) [16] is the PoA algorithm implemented in Parity, the Rust-based Ethereum client. The network is assumed to be synchronous and all authorities to be synchronised within the same UNIX time t. The index s of each step is deterministically computed by each authority as $s = t/step\_duration$, where $step\_duration$ is a constant determining the duration of a step. The leader of a step s is the authority identified by the id $l = s \ mod \ N$.

Authorities maintain two queues locally, one for transactions Q_{txn} and one for pending blocks Q_b. Each issued transaction is collected by authorities in Q_{txn}. For each step, the leader l includes the transactions in Q_{txn} in a block b, and broadcasts it to the other authorities (*block proposal* round in Figure 10.1(a)). Then each authority sends the received block to the others (round *block acceptance*). If it turns out that all the authorities received the same block b, then they *accept* b by enqueuing it in Q_b. Any received block sent by an authority not expected to be the current leader is rejected. The leader is always expected to send a block, if no transaction is available then an empty block has to be sent.

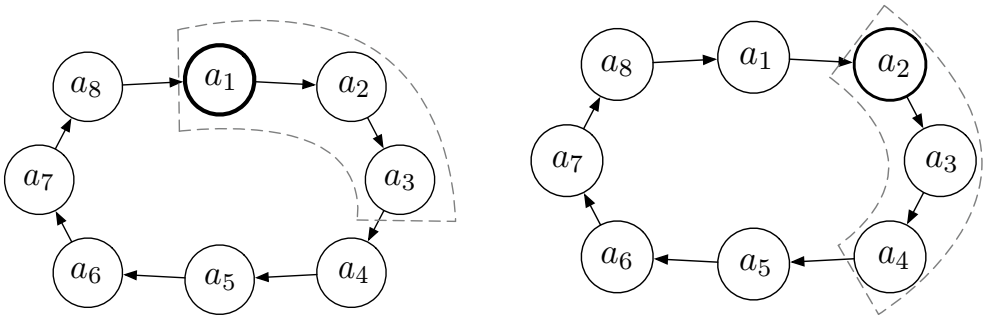
If authorities do not agree on the proposed block during the block acceptance, a voting is triggered to decided whether to kick off the leader. An authority can vote out the leader of current step because (i) it has not proposed any block, (ii) it has proposed more blocks, or (iii) it proposed different blocks to different authorities. The voting mechanism is realised through a *smart contract*, and a majority of votes is required to actually remove the current leader l from the set of legitimate authorities. When this happens, all the blocks in Q_b proposed by l are discarded. Note that leader

misbehaviours can be due to either benign faults (e.g., network asynchrony, software crash) or Byzantine faults (e.g., the leader has been subverted and behaves maliciously on purpose).

A block b remains in Q_b until a majority of authorities propose their blocks, then b is committed to the blockchain. With a majority of honest authorities, this mechanism should prevent any minority of (even consecutive) Byzantine leaders to commit a block they propose. Indeed any suspicious behaviour (e.g., a leader proposes different blocks to different authorities) triggers a voting where the honest majority can kick the current leader out, and the blocks they have proposed can be discarded before being committed.

10.2.2 Clique

Clique [53] is the PoA algorithm implemented in Geth, the golang-based Ethereum client. The algorithm proceeds in *epochs*, identified by a prefixed sequence of committed blocks. When a new epoch starts, a special *transition block* is dispatched through the network. It specifies the set of authorities (i.e., their *id*s) and can be used as snapshot of the current blockchain by new authorities needing to synchronise.



(a) Time t1 (b) Time t2

Figure 10.2: *Selection of authorities allowed to propose blocks in Clique.*

Conversely to Aura, which relies on the UNIX time to elect the leader, in Clique the *step* and the related leader are computed through a formula combining the block number and the number of authorities. Besides the leader, other authorities are allowed to propose blocks in a same step. To avoid that a Byzantine authority can wreck havoc the network by imposing too many blocks, each authority is allowed to propose at most one block every $N/2+1$ consecutive blocks. Thus, at any point in time there are $N-(N/2+1)$ authorities allowed to propose a block. If authorities act maliciously (e.g., by proposing a block when not allowed) they can be voted out. Specifically, at each step a vote against an authority can be casted. If a majority votes an authority out, the latter is removed from the list of legitimate authorities.

Having more authority that can propose a block during a same step, forks can occur. To reduce the probability of forks in a step, each non-leader authority allowed to propose a block delays the proposal by a random time, in order to give the block of the leader more chances to be the first received by all the authorities. Furthermore, a higher *score* is assigned to blocks proposed

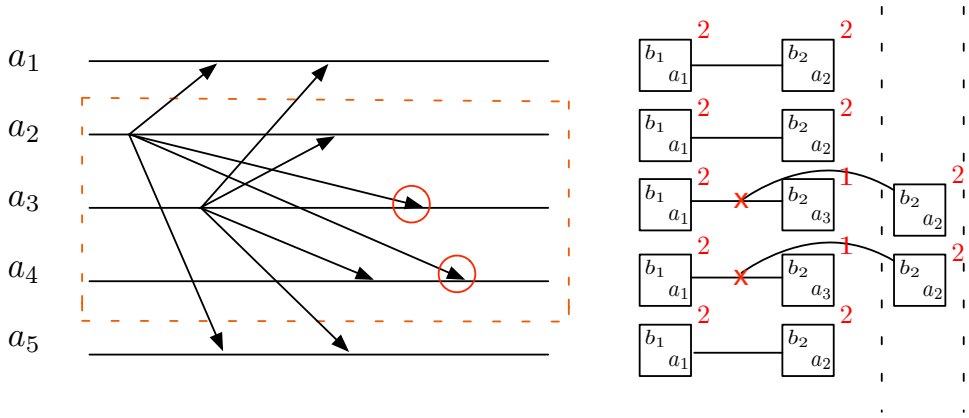


Figure 10.3: *A fork occurring in Clique. Authority a_4 has the block proposed by a_3 as second block, while a_5 has the block proposed by a_2. Eventually, a_4 replaces the block proposed by a_3 with that proposed by a_2 because the latter has a higher score.*

by the leader to prioritise its block in case of forks. Figure 10.2 shows two consecutive steps in which the designated leader changes as well as the set of authorities allowed to propose. In this example there are $N = 8$ authorities. At any step there are $N - (N/2 + 1) = 3$ authorities allowed to propose a block. One of them is the expected leader (the bold node in Figure 10.2). In Figure 10.2(a), a_1 is the expected leader, a_2 and a_3 are allowed to propose. In Figure 10.2(b), a_1 proposed at the previous step, so it has to wait other new $N/2 + 1$ blocks, a_4 is authorised to propose and a_2 is the new leader.

Figure 10.1(b) shows that at each step the leader broadcasts a block and all the authorities simply commit the proposal. Figure 10.3 depicts an example of a step in which a_2 is the designated leader and a_3, a_4 are allowed to propose a block. In such a scenario, a_2 proposes a block which arrives to a_1 and a_5. With a certain delay, a_3 proposes its block, which arrives to a_4 before the block proposed by a_2. Thus a fork occurs, as shown in the right part of Figure 10.3. This kind of forks can be easily detected by authorities in isolation when they receive a new block: if it does not reference the expected block (i.e., the last block of the local chain), then a fork exists. By relying on the GHOST protocol [234], forks can be eventually solved. Specifically, the authorities converge to the chain with the highest score, i.e. the chain having most blocks proposed by the designated leaders.

10.3 Comparison of Aura, Clique and PBFT

Previous PoA algorithms are here compared with PBFT, first in terms of consistency and availability properties via the CAP Theorem (Section 10.3.1), then of performance (Section 10.3.2).

10.3.1 Consistency and Availability Analysis based on CAP Theorem

The CAP Theorem [85, 37, 96] states that in a distributed data store only two out of the three following properties can be ensured: *Consistency* (C), *Availability* (A) and *Partition Tolerance* (P). Thus any distributed data store can be characterised on the basis of the (at most) two properties it can guarantee, either CA, CP or AP. Before delving into the CAP-based analysis of the considered algorithms, we refine the definitions of the three properties in the context of permissioned blockchains (deployed over the Internet). Notice also that WAN deployments such these are target of unforeseeable network delays of variable durations. In the following, we therefore assume an *eventually synchronous network model* where messages can be delayed among correct nodes, but eventually the network starts behaving synchronously and messages will be delivered (within a fixed but unknown time bound). This model is considered appropriate when designing real resilient distributed systems [42].

Consistency. A blockchain achieves consistency when forks are avoided. This property, as reported in Section 7.1.2, is referred to as *consensus finality* [229] which, in the standard distributed system jargon, corresponds to achieving the *total order* and *agreement* properties of *atomic broadcast*. The latter is the communication primitive considered as the type of consensus relevant for blockchains [42]. When consistency cannot be obtained, we have to distinguish whether forks are resolved sooner or later (*eventual consistency*) or they are not (*no consistency*).

Availability. A blockchain is available if transactions submitted by clients are served and eventually committed, i.e., permanently added to the chain.

Partition Tolerance. When a network partition occurs, authorities are divided into disjoint groups in such a way that nodes part of different groups cannot communicate each other. Internet-deployed permissioned blockchains also require the capability to tolerate the following adverse situations: (i) periods where the network behaves asynchronously; (ii) up to a certain number of authorities are Byzantine and act maliciously, i.e. aiming at preventing availability and consistency achievement. The maximum number of tolerable Byzantine nodes derives from the claims of each algorithm: any minority for PoA, less than one third for PBFT.

In the considered scenario, forfeiting tolerance to the reported situations becomes an unrealistic option, i.e. CA blockchains are not taken into account. Rather, we analyse the algorithms by assessing whether they keep the blockchain either consistent or available when such situations arise, i.e., whether the blockchain is either CP or AP (see Figure 10.4).

Aura Analysis

Authorities' clocks can drift and become out-of-synch. When authorities are distributed geographically over a really wide area, resynchronisation procedures cannot be perfectly effective because of network eventual synchrony. Hence, there can be periods where authorities do not agree on what is the current step, and thus on which authority is the current leader. Clocks' skews can be reasonably assumed strictly lower than the step duration, which is in the order of seconds, thus

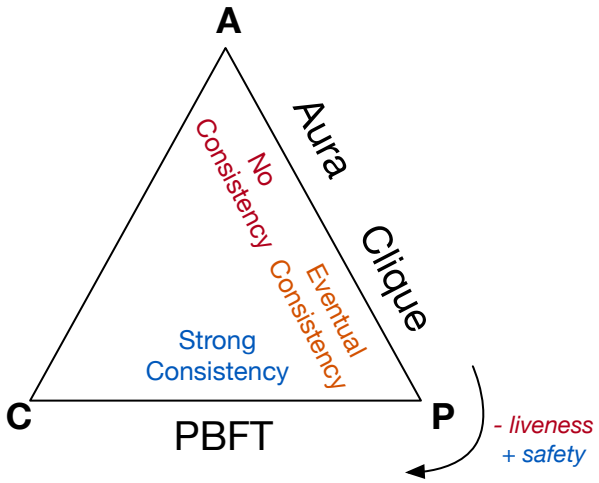


Figure 10.4: *Classification of Aura, Clique and PBFT according to the CAP Theorem*

we can have short time windows where two distinct authorities are considered as leader by two disjoint sets of authorities $\mathcal{A}_1$ and $\mathcal{A}_2$. Let $N_1 = |\mathcal{A}_1|$ and $N_2 = |\mathcal{A}_2|$, with $N_1 + N_2 = N$. By simply imposing an odd number of authorities, we can have a majority agreeing on who is the current leader. We can assume $N_1 > N_2$, i.e., $N_1 > N/2$ and $\mathcal{A}_1$ is a majority of authorities. Such a situation is depicted in Figure 10.5: authorities in $\mathcal{A}_1 = \{a_1, a_3, a_5\}$ see steps slightly out of phase with respect to the authorities in $\mathcal{A}_2 = \{a_2, a_4\}$. Indeed, the time windows coloured in grey are those where $\mathcal{A}_1$ disagrees with $\mathcal{A}_2$ on who is the current leader.

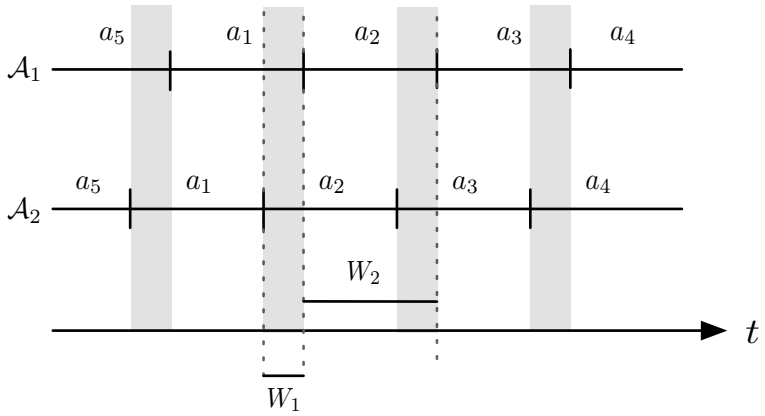


Figure 10.5: *Example of out-of-synch authorities in Aura (each step for a set of authorities is labelled by the expected leader)*

During time window W_1, a_2 considers itself the leader and sends a block to the other authorities. a_2 is believed to be the leader by the authorities in $\mathcal{A}_2$ but not by those in $\mathcal{A}_1$, hence the former accept its block while the latter reject it. During the time window W_2, authorities in $\mathcal{A}_1$ expect a_2 to send a block but this does not occur because it already sent its block for this step, thus at the end of W_2 they vote a_2 as malicious. Because a majority votes a_2 as malicious, it is removed by the set of authorities. All the other authorities in $\mathcal{A}_2$ are voted out one by one analogously, within

a number of steps lower than that required to achieve finality and commit the blocks, hence the consistency is preserved.

It is to note that in this case all the authorities are honest. Let us consider a scenario where there are B malicious authorities, they all are in $\mathcal{A}_1$ and do not vote authorities in $\mathcal{A}_2$ as Byzantine. If $B \geq N_1 - N/2$, then a majority is not reached to vote out authorities in $\mathcal{A}_2$, hence the blocks they propose and accept achieve finality and are committed to their local chains, causing a fork that is never resolved, i.e., we have *no consistency*. A minority of Byzantine authorities is enough to make this attack succeed.

Indeed, let us consider a set $\mathcal{S}$ with an odd number of elements $N = 2K + 1$, and a partition of such set in two non-empty subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ with cardinality N_1 and N_2, respectively, such that $\mathcal{S}_1$ is a majority and $\mathcal{S}_2$ a minority, i.e.,

$$K + 1 \leq N_1 \leq 2K \tag{10.1}$$

$$1 \leq N_2 \leq K$$

$$N_1 + N_2 = N$$

We want to prove that it suffices to remove a minority [2] of B elements from $\mathcal{S}_1$ to make it become a minority. Hence, we want to prove that

$$\exists B \mid N_1 - B \leq K \wedge B \leq K \tag{10.2}$$

Proof. Equation 10.2 can be proved by demonstrating that $N_1 - K \leq K$. This expression can be written as $N_1 \leq 2K$, which is always verified because of Equation 10.1.

Anyway, transactions keep being committed over time regardless of what a minority of Byzantine authorities do. In conclusion, Aura can be classified as an AP system according to the CAP theorem.

Clique Analysis

By design, Clique allows more authorities to propose a block with random delays, in order to cope with leaders that may not send any block because of network asynchrony, software crashes or malicious behaviours. Hence, forks are expected to occur with a certain probability. Clique, by relying on the Ethereum GHOST protocol, eventually reconciles these forks to achieve consensus on a single chain (i.e. we have *eventual consistency*). The minting frequency of authorities is bounded by $\frac{1}{N/2+1}$, therefore a majority of Byzantine authorities is required to take over the blockchain. This PoA algorithm can thus be classified as AP according to the CAP theorem.

[2] A minority with respect to the set $\mathcal{S}$.

PBFT Analysis

As long as less than one third of nodes are Byzantine, PBFT has been proved to guarantee consistency, i.e., no fork can occur [47]. Because of the eventual synchrony of the network, the algorithm can stall and blocks cannot reach finality. In this case, consistency is preserved while availability is given up. PBFT can then be easily classified as a CP system according to the CAP theorem.

10.3.2 Performance Analysis

The analysis here reported is qualitative and only based on how the consensus algorithms work in terms of message exchanging. The performance metrics usually considered for consensus algorithms are transaction latency and throughput. In the specific case of permissioned blockchains, we measure the latency of a transaction t as the time between the submission of t by a client and the commit of the block including t. Contrary to CPU intensive consensus algorithms such as PoW, here we can safely assume that latency is communication-bound rather than CPU-bound, as there is no relevant computation involved. Hence, we can compare the algorithms in terms of the number of message rounds required before a block is committed. Evaluating the throughput at a qualitative level is much more challenging, as it closely depends on the specific parallelisation strategy (e.g., pipelining) employed by each algorithm implementation. Thus, we deem more correct to compare throughput performance by the means of proper experimental evaluations that we plan to carry out as future work.

We assess how many message rounds are required for each algorithm in the normal case, i.e. when no condition occurs that makes any corner case to be executed. For example, for Aura we do not consider the situation when some authorities suspect the presence of subverted nodes and trigger a voting.

In Aura, each block proposal requires two message rounds: in the first round the leader sends the proposed block to all the other authorities, in the second round each authority sends the received block to all the other authorities. A block is committed after that a majority of authorities proposed their blocks, hence the latency in terms of message rounds in Aura is $2(N/2+1)$, where N is the number of authorities.

In Clique, a block proposal consists of a single round, where the leader sends the new block to all the other authorities. The block is committed straight away, hence the latency in terms of message rounds in Clique is 1. Such a huge difference between Aura and Clique is due to their different strategies to cope with malicious authorities aiming at creating forks: while Aura waits that enough other blocks have been proposed before committing, Clique commits immediately and copes with possible forks after they occur. Clique seems to outperform PBFT too, which takes three message rounds to commit a block.

10.4 Discussion

In this chapter we derive the functioning of two prominent consensus algorithms for permissioned blockchains based on the PoA paradigm, namely Aura and Clique. We provide a qualitative comparison of them and PBFT in terms of consistency, availability and performance, by considering a deployment over the Internet where the network can be modelled more realistically as eventually synchronous rather than synchronous.

By applying the CAP Theorem, we claim that in this setting PoA algorithms can give up consistency for availability when considering the presence of Byzantine nodes. This can prove to be unacceptable in scenarios where the integrity of the list of transactions has to be absolutely kept (which is likely to be the actual reason why a blockchain-based solution is used). On the other hand, PBFT keeps the blockchain consistent at the cost of availability, even when the network behaves temporarily asynchronously and Byzantine nodes are present, which can be much more desirable when data integrity is a priority. Despite one of the most praised advantages of PoA algorithms consists in their performance, our qualitative analysis shows that in terms of latency the expected loss in PBFT is bounded, and can be offset by the gain in consistency guarantees.

As a future work, we aim to propose a benchmark to fairly compare consensus protocols of permissioned blockchain.

Part III

Final Outcome

Chapter 11

Conclusions

In this thesis we studied techniques to achieve and improve performability and dependability of complex modern distributed systems. The thesis has been structured in two parts. In the first part we targeted the *availability* through automatic scaling solutions, while in the second part we focused on the *security* of blockchain-based distributed architectures.

In Part I, we started with the claim that proactivity may lead to a more efficient usage of resources; so we designed MYSE, an architecture for automatic scaling of replicated services which combines workload forecasting and performance estimation. Although promising results, we assessed that it might be toughly employed in several real case scenario where performance estimation can be a tricky task. Thus, we refine the architecture by replacing the queuing model with a service profiler which collects metrics and learns system behaviour, thus we proposed PASCAL. We instantiate PASCAL with a solution for distributed datastore and we evaluated its efectiveness through an experimental evaluation of a prototype within Cassandra.

Similarly, we tried to instantiate PASCAL in a stream processing system context. However, in such environments the autoscaler needs to address two separate tasks: (i) scaling operators and (ii) scaling resources. Therefore, we proposed ELYSIUM, a solution for elastically scale both dimensions *symbiotically*, i.e. in an independent manner, yet in order to optimise the overall system efficiency. We evaluated ELYSIUM through an experimental evaluation conducted on a real prototype integrated within the Scheduler of Storm.

From both case studies on Cassandra and Storm we showed as a proactive approach with an adequate solution to estimate performances can outperform a reactive system and save more resources. Such results backup our first claim proposed in MYSE and confirm its simulation results.

In Part II, we introduced the following idea: leveraging blockchain to improve security of dependable distributed systems in presence of parties (or nodes) which may behave maliciously. We started with a solution to solve disputes in multi-party transactions by proposing SLAVE, an architecture which exploits a blockchain and asymmetric cryptography. This solution make us reason about employability of blockchain as a general purpose distributed database. We pro-

posed so the main open research questions to realise such a database to come up with two aspects which need to be balanced: performances and security. Indeed, typical blockchain (permissionless) achieve very poor performances, but they can ensure high data integrity. On the other hand, permissioned blockchain can achieve higher performances sacrificing some security guarantees. We proposed a solution to integrate both of them in a layered architecture, i.e. 2LBC. A permissioned blockchain is employed on the first layer as transaction log of the distributed database and periodically hashes of such blockchain are stored to a permissionless blockchain in the second layer (anchoring). We showed through a Java prototype anchored with Ethereum how the performance can be increased, and we analysed also that integrity can still be preserved with some extent.

Furthermore, being blockchain a fully decentralised technology, how involved parties agree on transaction ordering plays a key role for both security and performance perspectives. Hence we studied new democratic schemas to achieve agreement in such environments, addressing so one of the most prominence problem of distributed systems, namely, *distributed consensus*. We investigated so the new Proof-of-Authority consensus schema and compared through a CAP theorem-based analysis its effectiveness rather than a typical PBFT-based approach showing as consensus safety (and thus, blockchain consistency) may be compromised, leading to periods of forks which may make the blockchain not reliable.

Chapter 12

Future Directions

The main objective to carry on the work of this thesis is the integration of the two parts. The idea is to define an *elastic dependable framework* which allows to reconfigure runtime the system in a decentralised manner.

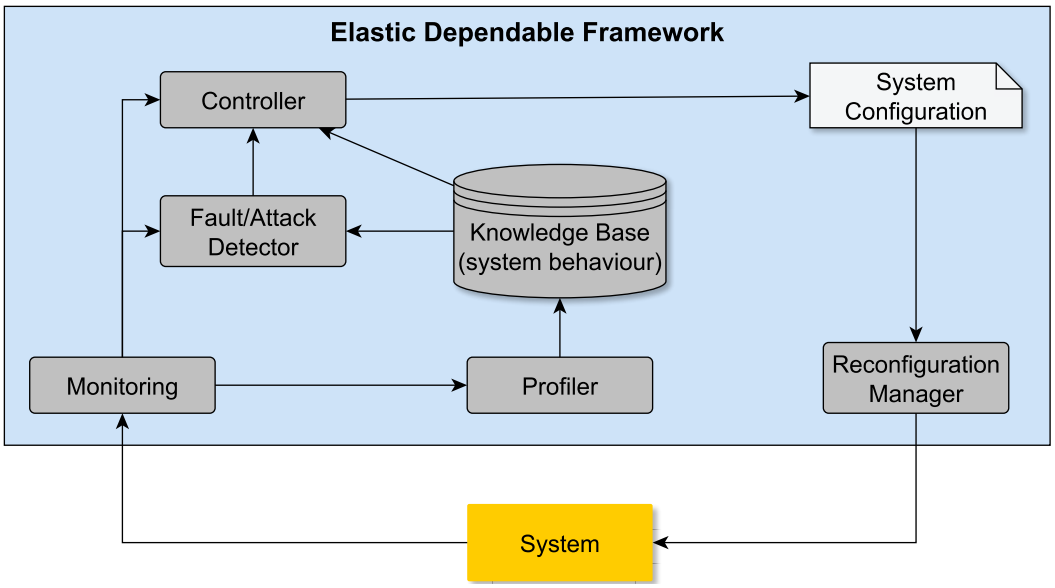


Figure 12.1: *Decentralised Elastic Dependable Framework*

Figure 12.1 shows a possible architecture of the elastic dependable framework. Ideally, it is deployed on top of a permissioned blockchain, used as underlying infrastructure, anchored with a permissionless blockchain as proposed for the 2LBC architecture. Compared to PASCAL, events triggering reconfigurations are not only workload changes, yet also faults and attacks, thus further modules beyond PASCAL are necessary. Specifically, a *fault/attack detector*[1], by revealing (or forecasting) a tough conditions such as a fault or an attack, can interact with the controller to properly reconfigure the system.

[1]Along this directions, my contributions are NiTREC [26] and NIRVANA [51], two failure detection/prediction systems which correlate observed metrics to infer whether the system is behaving correctly or there is a fault

Therefore, the membership of the permissioned network might be managed runtime through voting smart contracts. Therefore, subverted nodes which behave maliciously can be voted out (as we assessed with Proof-of-Authority algorithms in Chapter 10) to maintain a honest membership.

This framework, so, will be a decentralised extension of PASCAL in which decisions of reconfigurations are taken with the consensus of network members. For this scope, one of the main challenges moves along consensus protocols for permissioned blockchain. Indeed, solutions based on PBFT fails in performances and scalability, yet solution based on Proof-of-Authority seems to be too weak in consistency as they propose *eventual consensus* which can be not ideal for scenarios where forks may lead to huge harm. Therefore, in order to speed up the system while guaranteeing *safety* and *liveness* with Byzantine faults, will be investigated new protocols specific for permissioned blockchain.

For what concern instead the permissionless blockchain (anchored as second layer), being Proof-of-Work so heavy and cost ineffective, I plan to investigate some different model to encourage users to behave honestly. Currently, cryptocurrencies-based blockchain like Bitcoin and Ethereum employ monetary rewards for users who mine correctly blocks. Sociology community has widely proven as without an incentive, ethic will lose [106]. I propose so to replace the concept of *incentive* with the concept of *necessity*. Thereby, there can be found crucial factors which miners need to ”survive”; thus, game theory can be exploited to define a game in which a player is discouraged to behave maliciously rather than encouraged to behave honestly.

"Be a voice not an echo."

Albert Einstein

Appendices

Appendix A

Artificial Neural Network

Artificial Neural Networks (ANNs) are models inspired by the brain with a network of interconnected *Artificial Neurons* able to compute the output values by feeding information through the network. They are widely used for prediction and time-series forecasting [239].

A.1 Artificial Neuron

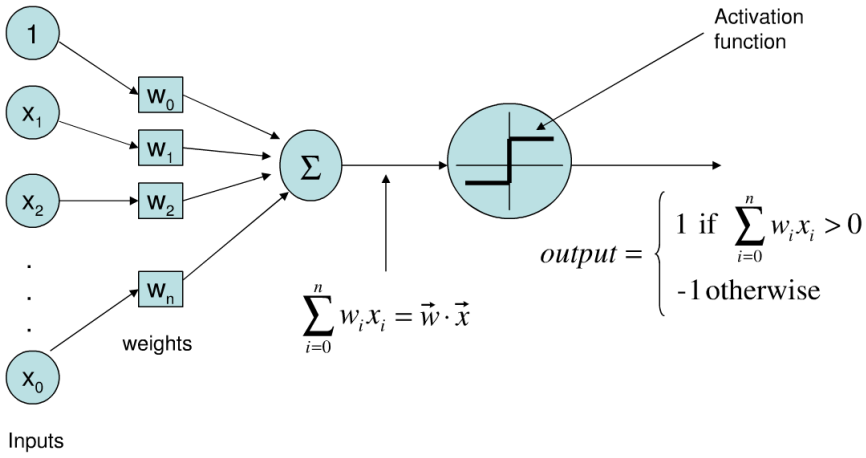


Figure A.1: *Neuron architecture*

An *Artificial Neuron* is a mathematical function introduced by McCulloch & Pitts in 1943 [131] which has $m+1$ inputs with signals x_0 trough x_m and weight w_0 trough w_m. Usually the first input x_0 is assigned to value $+1$ which makes it a *bias* input with $w_{k0} = b_k$, so only m actual inputs to the neuron.
The output of a generic neuron k is:

$$y_k = \phi(\sum_{j=0}^{m} w_{kj} x_j)$$

where ϕ is a transfer function which has a task of activation function *"ON-OFF"*. A typical transfer function is the *step function* where a threshold establish when a sum of signals is enough strong to go ahead $(+1)$ otherwise it has to stop (-1):

$$f(x) = \begin{cases} +1 & \text{if } w \cdot x + b > 0 \\ -1 & \text{otherwise} \end{cases}$$

Actually, that function is too much strongly selective, so there exist many different transfer functions belonging to sigmoid functions family, which can be smoother than the step function and to give as output a real number in a range.

A.2 The Perceptron

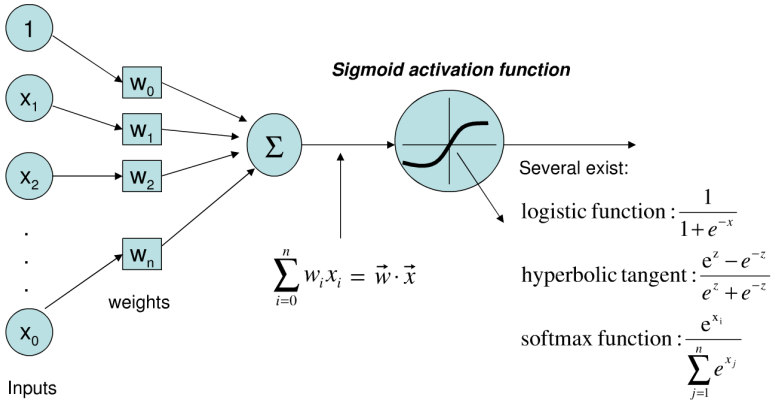


Figure A.2: *Perceptron architecture*

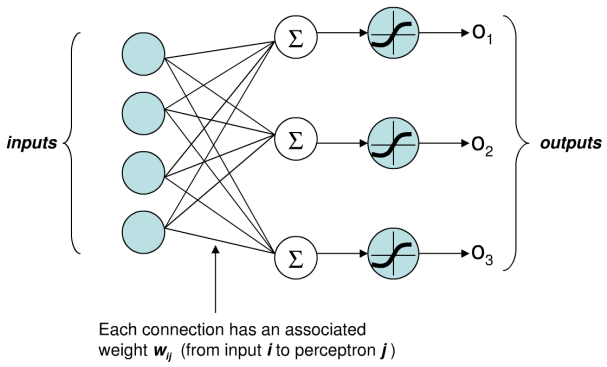


Figure A.3: *Single-Layer Perceptron architecture*

In Artificial Neuron Networking the *Perceptron*, introduced by Rosenblatt in 1962 [199], is the easiest data structure that improve the *Artificial Neuron* by using a smoother activation function.

The *Perceptron* can be trained with a simple supervised algorithm (Delta Rule [6]) that iteratively present from a *training set* the inputs to *Perceptron* which calculate the output and having the actual output value is able to compute the error and update the weights on every *Dendrites* which are the synapses i.e. the final part of links coming from previous layer neurons.

The *Single-Layer Perceptron* (SLP) is a type of ANN having an input layer with the input nodes, that are not neurons, as they do not make any computation, and an output layer with one or more output neurons. SLPs are able to learn some input and output patterns, but Minsky and Papert demonstrated as those have to be *linearly separable* [168], i.e. if in an n-dimensional space they are able to separate those by a hyperplane. More formally, let X_0 and X_1 be two sets of points in an n-dimensional space. Then, X_0 and X_1 are linearly separable if there exists $n+1$ real numbers $w_1, w_2, .., w_n, k$, such that every point $x \in X_0$ satisfies $\sum_{i=1}^{n} w_i x_i \geq k$ and every point $x \in X_1$ satisfies $\sum_{i=1}^{n} w_i x_i < k$, where x_i is the $i-th$ component of x.

A.3 Multi-Layer Perceptron

Having only one layer, there not exist a way for SLP to learn patterns which is not *linearly separable*. A multi-layered network overcomes this limitation as it can create internal representations and learn different features in each layer.

The most common used ANN are the *Feed-Forward* whose interconnections among *Perceptrons* follow only a direction towards the next layers fully connected, but there exist other connection types as *Recurrent*. A *Multi-Layer Perceptron*, has an *input layer*, an *output layer* and one or more *hidden layers* made by an arbitrary number of neurons (or perceptrons).

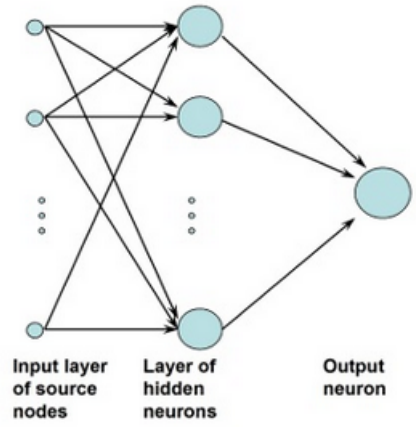


Figure A.4: *Multi-Layer Perceptron architecture with 1 hidden layer*

A.4 Learning from Data

A *Multi-Layer Perceptron* (MLP) can be trained so as to give it the generalization power, i.e. the capability to predict unseen values. The main approaches for learning is by employing a *Back-Propagation Algorithms* (*backprop*), i.e. a *supervised learning* algorithms based on computing and backpropagating the output error and gradient descent. Other approaches are less common, but used in some field, specifically are used *Genetics Algorithms* (GAs).

A.4.1 Backpropagation Algorithm

This algorithm was introduced by Rumelhart, Hinton & Williams in 1985 [201] and it is the generalisation to a MLP of *Delta Rule* which is a rule for updating the network weights on a SLP based on *gradient descent*. *Gradient Descent* (also called *steepest descent*) is a first-order optimisation algorithm useful to find a local minimum of a function by using gradient descent by taking steps proportional to the negative of the function gradient at the current point.

The aim of *backprop* is to works by decreasing iteratively the error produced of ANN on the output nodes than the expected result and by backpropagating such error trough the network to update the weights of the *dendrites* of every *perceptrons*.

We represent the error in output node j in the n_{th} data point by $e_j(n) = d_j(n) - y_j(n)$, where d is the target value and y is the value produced by the *perceptron*. We then make corrections to the weights of the nodes based on those corrections which minimise the error in the entire output, given by

$$\mathcal{E}(n) = \frac{1}{n}\sum_j e_j^2(n)$$

Using gradient descent, we find our change in each weight to be

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

where y_i is the output of the previous neuron and η is the *learning rate*, which is carefully selected to ensure that the weights converge to a response quickly, without producing oscillations. In programming applications, this parameter typically ranges from 0.2 to 0.8.The derivative is depending on the induced local field v_j, which itself varies. It pretty easy to prove that for an output node this derivative could be simplified to

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n))$$

where ϕ' is the derivative of the activation function described earlier, which itself does not vary. The same analysis is harder to do for the change in weights to hidden nodes, but it can be shown that the relevant derivative is

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial v_k(n)} w_{kj}(n)$$

This depends on the change in weights of the k_{th} nodes, which represent the output layer. So to change the hidden layer weights, we must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a *backprop* of the activation function.

After calculating the change of weights, the algorithm calculates the update of every *Dendrite* weights:

$$\Delta w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n) + \underbrace{\beta * \Delta w_{ji}(n-1)}_{for\ improved\ \beta\ version}$$

where β is a *momentum* term and its useful is for helping the learning process. We focus this aspect below.

There exist 2 main versions of *backprop*:

1. *Online*: means a weight update after each pattern.
   - order in which learning samples are presented plays a role
   - usually more accurate than *offline*
2. *Offline (batch learning)*: is a variant in which the weights variations are accumulated during a training epoch, and the network weights are actually updated at the end of a training epoch.
   - order in which learning samples are presented does not play a role
   - more memory consumption needed
   - faster than *online*

The worst limitations of *backprop* are:

- Local Minima suffering: i.e. going on with iterations the error rate should decrease but we may have fluctuations (Figure A.5). The *gradient descent* algorithms suffer from problem 1 of *local minima*, anyway there exist different ways to reduce this problem:
  1. using *Genetic Algorithms*
  2. choosing the *online* version of *backprop* algorithm: due to the noise introduced in the error surface by online learning, it can by itself be sufficient to avoid local minima, but because of the random oscillation introduced, online learning is usually slower that offline learning.
  3. using the *momentum* term which help the algorithm to converge

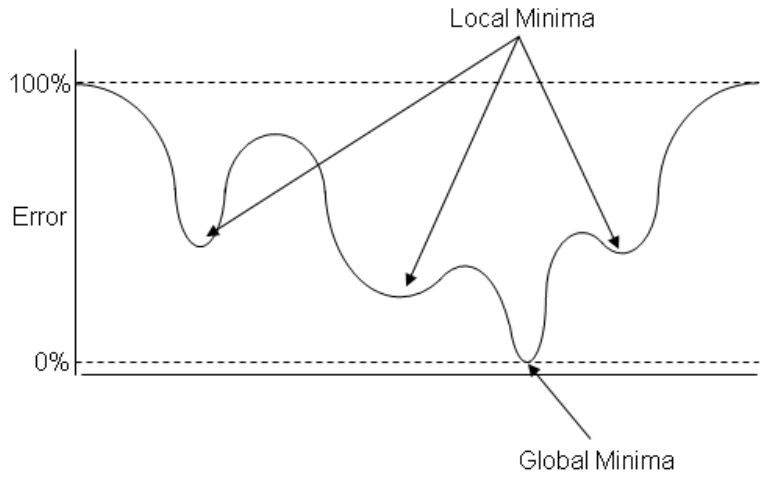


Figure A.5: *Local Minima problem*

- Slow convergence to optimum
- Convergence to optimum not guaranteed

To improve the performance of the ANN and to reduce the convergence problems *backprop* can use, even is not required, a normalisation of input vectors [38].

More, there exist different improved versions of *backprop* which are aimed to reduce such limitations [228, 225, 121, 231] and one of more used version is the *Resilient Backprop*.

A.4.2 Resilient Backpropagation Algorithm

This algorithm has been introduced by Riedmiller & Braun in 1992 [196] and it is a batch update version of backpropagation. There exist several variants, and it is considered the best version of *backprop* right now [122, 137].

The algorithm uses the same methods of traditional Backprop for the calculation of errors and for the calculation of partial derivatives of each neurons, but the purpose of this algorithm, also called RPROP, is to eliminate the harmful effects of the magnitudes of the partial derivatives by using only the sign of the derivative to determine the direction of the weight update.

The algorithm uses an update value Δ_{ij} for each synaptic weights and the reason of using the derivative sign is due to the fact that when the derivative changes its sign compared to the previous step, means the algorithm has just jumped over a minimum and the update value was to high. In such case RPROP decreases the update value, in order to have at the next step a jump less than the previous and the descent go on towards the minimum. If instead the derivative maintain the same sign the update value is incremented in order to speed up the convergence.

$$\Delta_{ij}(t) = \begin{cases} \eta^+ * \Delta_{ij}(t-1), & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}} * \frac{\delta E(t-1)}{\delta w_{ij}} > 0 \\ \eta^- * \Delta_{ij}(t-1), & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}} * \frac{\delta E(t-1)}{\delta w_{ij}} < 0 \\ \Delta_{ij}(t-1) & \text{otherwise} \end{cases}$$

where η^+ and η^- are respectively the positive and negative step value typically set up to 1.2 and 0.5. After the setting of the update value Δ_{ij} the synaptic weights update follows the rule below:

$$\Delta W_{ij}(t) \begin{cases} -\Delta_{ij}(t), & \text{if } \frac{\delta E(t)}{\delta w_{ij}} > 0 \\ +\Delta_{ij}(t), & \text{if } \frac{\delta E(t)}{\delta w_{ij}} < 0 \\ 0 & \text{otherwise} \end{cases}$$

Appendix B

Q-Learning

Q-Learning is a model-free reinforcement learning technique [230]. Specifically, the Q-Learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process. It works by learning an action-value function that gives the expected benefit of taking a given action in a given state and thereafter following the optimal policy. A *policy* is a rule that the agent follows to select the actions, given its current state. When such an action-value function is learned, the optimal policy is constructed simply by selecting for each state, the action with the highest value.

The problem model consists of an agent, states S and a set of actions per state A. By performing an action $a \in A$, the agent can move from a state to another. Executing an action in a specific state provides the agent with a reward (a numerical score). The goal of the agent is to maximise its total reward. It does this by learning which action is optimal for each state.

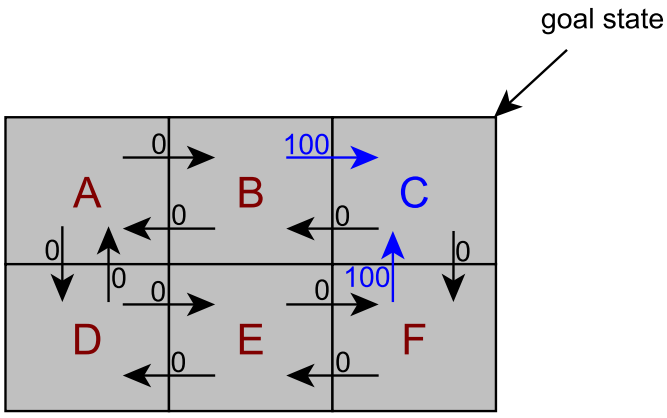


Figure B.1: *States and reward related to actions*

The action that is optimal for each state is the action with the highest long-term reward. This reward is a weighted sum of the expectation values of the rewards of all future steps starting from

the current state, where the weight for a step from a state Δt steps into the future is calculated as $\gamma^{\Delta t}$. Here, γ is a number between 0 and 1 ($0 \leq \gamma \leq 1$) called the discount factor and trades off the importance of sooner versus later rewards. Specifically, a discount factor equal to 0 makes the agent *myopic*, thus it considers only current rewards, while a discount factor equal or grater to 1 the action can lead to very long training as it consider also past values.

The algorithm therefore has a function that calculates the Quantity of a state-action combination: $Q : S \times A \rightarrow \mathbb{R}$.

Before starting to learn Q returns an (arbitrary) fixed value, chosen at the beginning. The initial condition Q_0 plays a key role and high initial values may encourage the exploration. Several approach to cope with the initial condition problem have been proposed as [209] for resetting such a parameter. Once the learning starts, at each iteration the agent selects an action, observes the reward and a new state that may depend on both the previous state and the selected action, Q is updated. The core of the algorithm is a simple value iteration update. It assumes the old value and makes a correction considering the new information.

$$Q_{t+1}(s_t, a_t) = \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{opt future val}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right) \tag{B.1}$$

where R_{t+1} is the reward observed after performing a_t in s_t, and where $\alpha_t(s, a)(0 < \alpha \leq 1)$ is the learning rate (may be the same for all pairs).

Specifically, the reward can follow a specific function which assign a value to each action bringing from a state to another; for instance, it can assign a maximum value if the action brings to the goal state, zero otherwise, as the example in Figure B.1 shows.

The learning rate, instead, is a parameter which determines how the new information overwrite the current ones, i.e. a learning rate equal to 0 makes the agent not sensible to new information and thus it will not learn anything, while a learning rate equal to 1 makes the agent so sensible to new information to consider only the most recent ones.

List of Tables

List of Figures

List of Acronyms

2LBC	Two Layer Blockchain
3PC	Three-Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
ANN	Artificial Neural Network
Aura	Authority Round
AWT	Approximated Workload Trace
B2B	Business-to-Business
BASE	Basically Available, Soft state, Eventual consistency
BFT	Byzantine Fault Tolerance
CAP	Consistency, Availability, Partition tolerance
DoS	Denial-of-Service
DP	Demand Point
DPoS	Delegated Proof-of-Stake
ELYSIUM	Elastic Symbiotic Scaling of Operators and Resources in SPS
FaaS	Federation-as-a-Service
IaaS	Infrastructure-as-a-Service
ILP	Input Load Profiler
MEF	Ministry of Economy and Finance
MIN	Ministry of Interior
MITA	Malta Information Technology Agency
MLP	Multi Layer Perceptron
MYSE	Make Your System Elastic
NT-MYSE	Non Trained-MYSE
OCUP	Operator CPU Usage Profiler
OP	Overhead Profiler
OPP	Over-provisioning Penalty
PaaS	Platform-as-a-Service
PASCAL	Proactive Automatic Scaling
PBFT	Practical Byzantine Fault Tolerance

PoA	Proof-of-Authority
PoB	Proof-of-Burn
PoC	Proof-of-Capacity
PoE	Proof-of-Existence
PoET	Proof-of-Elapsed Time
PoI	Proof-of-Importance
PoS	Proof-of-Stake
PoV	Proof-of-Validation
PoW	Proof-of-Work
QoS	Quality-of-Service
RP	Reconfiguration Point
SaaS	Software-as-a-Service
SASO	Stability, Accuracy, Short settling time, no Overhead
SCP	Stellar Consensus Protocol
SL	Service Ledger
SLI	Service Ledger Interface
SLA	Service Level Agreement
SLAVE	Service Level Agreement Verified
SMR	State Machine Replication
SP	Selectivity Profiler
SPS	Stream Processing System
TP	Triggering Point
TPE	Total Penalty
TTP	Trusted Third Party
UPP	Over-provisioning Penalty
VM	Virtual Machine

Bibliography

[1] Microsoft Office 365. `https://outlook.office365.com/`.

[2] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[3] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, page 7. ACM, 2013.

[4] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212, 2012.

[5] Arnold O Allen. *Probability, statistics, and queueing theory*. Academic Press, 2014.

[6] John Robert Anderson, Ryszard Spencer Michalski, Ryszard Stanisław Michalski, Thomas Michael Mitchell, et al. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[7] L. Aniello, R. Baldoni, E. Gaetani, F. Lombardi, A. Margheri, and V. Sassone. A prototype evaluation of a tamper-resistant high performance blockchain-based transaction log for a distributed database. In *EDCC*. IEEE, 2017.

[8] Leonardo Aniello, Roberto Baldoni, and Federico Lombardi. A blockchain-based solution for enabling log-based resolution of disputes in multi-party transactions. In *5th International Conference on Software Engineering for Security and Defense Applications*. Springer, 2016.

[9] Leonardo Aniello, Silvia Bonomi, Federico Lombardi, Alessandro Zelli, and Roberto Baldoni. An architecture for automatic scaling of replicated services. In *NETYS '14*, pages 122–137. Springer, 2014.

[10] Apache Flink. `https://flink.apache.org/`.

[11] Karen Appleby, Sameh Fakhouri, Liana Fong, Germán Goldszmidt, Michael Kalantar, Srirama Krishnakumar, Donald P Pazel, John Pershing, and Benny Rochwerger. Oceano-sla based management of a computing utility. In *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 855–868. IEEE, 2001.

[12] Google Apps. `https://apps.google.com/`.

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[14] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[15] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. ACM, 2007.

[16] Aura. https://github.com/paritytech/parity/wiki/Aura.

[17] Traces available in the Internet Traffic Archive. `http://ita.ee.lbl.gov/html/traces.html`.

[18] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[19] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: a taxonomy. *Building the Information Society*, pages 91–120, 2004.

[20] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[21] Amazon AWS. `https://aws.amazon.com/`.

[22] Microsoft Azure. `https://azure.microsoft.com/`.

[23] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB*, 13(4):333–353, 2004.

[24] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 2014.

[25] Arshdeep Bahga and Vijay Krishna Madisetti. Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, 4(07):396, 2011.

[26] Roberto Baldoni, Adriano Cerocchi, Claudio Ciccotelli, Alessandro Donno, Federico Lombardi, and Luca Montanari. Towards a non-intrusive recognition of anomalous system behavior in data centers. In *International Conference on Computer Safety, Reliability, and Security*, pages 350–359. Springer, 2014.

[27] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. Shuttledb: Database-aware elasticity in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, 2014.

[28] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.

[29] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, David A Patterson, et al. Rain: A workload generation toolkit for cloud computing applications. *University of California, Tech. Rep. UCB/EECS-2010-14*, 2010.

[30] BFT-SMaRT. https://github.com/bft-smart/library/.

[31] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. Sla management in federated environments. *Computer Networks*, 35(1):5–24, 2001.

[32] Sangeeta Biswas, Shamim Ahmad, M. Khademul Islam Molla, Keikichi Hirose, and Mohammed Nasser. Kolmogorov-smirnov test in text-dependent automatic speaker identification. *Engineering Letters*, 16(4):469–472, 2008.

[33] IBM Bluemix. `https://www.ibm.com/cloud-computing/bluemix/`.

[34] Peter Bodík, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[35] André B Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203. ACM, 2000.

[36] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE, 2015.

[37] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[38] Mat Buckland. *A1 techniques for game programming*. CengageBrain. com, 2002.

[39] Rajkumar Buyya, James Broberg, and Andrzej M Goscinski. *Cloud computing: Principles and paradigms*, volume 87. John Wiley & Sons, 2010.

[40] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[41] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming*. Springer, 2011.

[42] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[43] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[44] M. Cardosa and A. Chandra. Resource bundles: Using aggregation for statistical large-scale resource discovery and management. *Parallel and Distributed Systems, IEEE Transactions on*, 21(8):1089–1102, 2010.

[45] Eddie Caron, Fréedéeric Desprez, and Adrian Muresan. Forecasting for Cloud Computing On-demand Resources Based on Pattern Matching. Research RR-7217, Inria, 2010.

[46] Apache Cassandra. http://cassandra.apache.org/.

[47] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[48] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD '13*, pages 725–736. ACM, 2013.

[49] SUNFISH KeyValueStore Chaincode. `https://github.com/sunfish-prj/Service-Ledger/blob/master/server/hyperledger-fabric/chaincode/keyvaluestore/key_value_store.go`.

[50] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 337–350, Berkeley, CA, USA, 2008. USENIX Association.

[51] Claudio Ciccotelli, Leonardo Aniello, Federico Lombardi, Luca Montanari, Leonardo Querzoni, and Roberto Baldoni. Nirvana: A non-intrusive black-box monitoring framework for rack-level fault detection. In *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pages 11–20. IEEE, 2015.

[52] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[53] Clique. https://github.com/ethereum/EIPs/issues/225.

[54] Google Cloud. `https://cloud.google.com/`.

[55] IBM Cloud. `https://www.ibm.com/cloud-computing`.

[56] CloudSimEx Cloudslab. `https://github.com/Cloudslab/CloudSimEx`.

[57] Nxt community. Nxt whitepaper. 2014.

[58] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[59] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[60] Couchbase. http://https://www.couchbase.com/.

[61] Transaction Processing Performance Council. Transaction processing performance council. *Web Site, http://www. tpc. org*, 2005.

[62] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[63] Balázs Csanád Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[64] George Danezis and Sarah Meiklejohn. Centrally Banked Cryptocurrencies. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[65] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal. Elastras: An elastic transactional data store in the cloud. *HotCloud*, 9:131–142, 2009.

[66] Apache Derby DB. https://db.apache.org/derby/.

[67] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[68] Nuno Diegues and Paolo Romano. Sti-bt: A scalable transactional index. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2408–2421, 2016.

[69] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *SIGMOD*, pages 1085–1100. ACM, 2017.

[70] Docker. `https://www.docker.com/`.

[71] SUNFISH Platform Documentation. `http://sunfish-platform-documentation.readthedocs.io/en/latest/`.

[72] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From Data Center Resource Allocation to Control Theory and Back. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 410–417, 2010.

[73] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74, Venice/Mestre, Italy, 2011.

[74] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[75] Amazon AWS EC2. `http://aws.amazon.com/ec2`.

[76] T Edwards, D Tansley, R Frank, and N Davey. Traffic trends analysis using neural networks. In *Procs of the Int Workshop on Applications of Neural Networks to Telecommunications*, 1997.

[77] Aaron J Elmore, Sudipto Das, Alexander Pucher, Divyakant Agrawal, Amr El Abbadi, and Xifeng Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2013.

[78] Google App Engine. `https://cloud.google.com/appengine`.

[79] Google Compute Engine. `https://cloud.google.com/compute`.

[80] ENISA. Security Framework for Governmental Clouds, 2015. Available at `https://www.enisa.europa.eu/activities/Resilience-and-CIIP/cloud-computing/governmental-cloud-security/security-framework-for-govenmental-clouds`.

[81] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, 2016.

[82] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[83] Hyperledger Fabric. `https://hyperledger.org/projects/fabric`.

[84] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[85] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 174–178. IEEE, 1999.

[86] ClarkNet HTTP Trace. (from the internet traffic archive). `http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html`.

[87] World Cup 98 Trace. (from the internet traffic archive). `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[88] A Michael Froomkin. The essential role of trusted third parties in electronic commerce. *Or. L. Rev.*, 75:49, 1996.

[89] Edoardo Gaetani, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Blockchain-based database to ensure data integrity in cloud computing environments. In *ITA-SEC*, volume 1816. CEUR-WS.org, 2017.

[90] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *The Bitcoin Backbone Protocol: Analysis and Applications*, pages 281–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[91] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting dependable systems*, pages 61–89. Springer, 2003.

[92] B. Gedik, S. Schneider, M. Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. In *Transactions on Parallel and Distributed Systems*, volume 25, pages 1447–1463. IEEE, 2014.

[93] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[94] Faban Workload Generator. `http://faban.org/`.

[95] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 173–178, New York, NY, USA, 2012. ACM.

[96] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[97] Inigo Goiri, Jordi Guitart, and Jordi Torres. Characterizing cloud federation for enhancing providers' profit. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 123–130. IEEE, 2010.

[98] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, 2010.

[99] BitFury Group. Incentive mechanisms for securing the bitcoin blockchain. *White Paper*, 2015.

[100] BitFury Group and Jeff Garzik. Public versus private blockchains part 1: Permissioned blockchains. *White Paper*, 2015.

[101] BitFury Group and Jeff Garzik. Public versus private blockchains part 2: Permissionless blockchains. *White Paper*, 2015.

[102] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. In *Transaction on Parallel Distributed System*, volume 23, pages 2351–2365. IEEE, 2012.

[103] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., 1993.

[104] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[105] Rui Han, Li Guo, M.M. Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 644–651, 2012.

[106] Garrett Hardin. The tragedy of the commons. *Journal of Natural Resources Policy Research*, 1(3):243–253, 2009.

[107] M.Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S.L.D. Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1327–1334, 2012.

[108] Heaton Research. Encog Machine Learning Framework. http://www.heatonresearch.com/encog/.

[109] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. Cloud-based data stream processing. In *DEBS '14*, pages 238–245. ACM, 2014.

[110] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *DEBS '14*, pages 13–22. ACM, 2014.

[111] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *DEBS '14*, pages 318–321. ACM, 2014.

[112] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. Online parameter optimization for elastic data stream processing. In *SoCC '15*, pages 276–287. ACM, 2015.

[113] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, volume 2007, pages 132–141, 2007.

[114] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[115] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, volume 13, pages 23–27, 2013.

[116] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. In *Computing Survey*, volume 46. ACM, 2014.

[117] Giles Hogben and Alain Pannetrat. Mutant apples: a critical examination of cloud sla availability definitions. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 379–386. IEEE, 2013.

[118] JBoss HornetQ. http://hornetq.jboss.org/.

[119] Chao-Wen Huang, Wan-Hsun Hu, Chia Chun Shih, Bo-Ting Lin, and Chien-Wei Cheng. The improvement of auto-scaling mechanism for distributed database-a case study for mongodb. In *APNOMS*, pages 1–3, 2013.

[120] Jinhui Huang, Chunlin Li, and Jie Yu. Resource prediction based on double exponential smoothing in cloud computing. In *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pages 2056–2060, 2012.

[121] DR Hush and JM Salas. Improving the learning rate of back-propagation with the gradient reuse algorithm. In *Neural Networks, 1988., IEEE International Conference on*, pages 441–447. IEEE, 1988.

[122] Christian Igel and Michael Hüsken. Empirical evaluation of the improved rprop learning algorithms. *Neurocomputing*, 50:105–123, 2003.

[123] Martin Illecker. SentiStorm. https://github.com/millecker/senti-storm.

[124] Infinispan. http://infinispan.org/.

[125] SUNFISH Service Ledger Interface. `https://github.com/sunfish-prj/Service-Ledger-Interface`.

[126] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.*, 27(6):871–879, June 2011.

[127] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *CLOUD '11*, pages 195–202. IEEE, 2011.

[128] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28(1):155–162, January 2012.

[129] Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Buğra Gedik, Henrique Andrade, Kun-Lung Wu, and Ravishankar K Iyer. Modeling stream processing applications for dependability evaluation. In *DSN '11*, pages 430–441. IEEE, 2011.

[130] Apache JMeter. `http://jmeter.apache.org/`.

[131] Dušan Joksimović. Neural networks and geographic information systems. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[132] Apache Kafka. http://kafka.apache.org/.

[133] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, Mike Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *OSDI*, volume 12, pages 237–250, 2012.

[134] Pradeeban Kathiravelu and Luis Veiga. An adaptive distributed simulator for cloud and mapreduce algorithms and architectures. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 79–88. IEEE, 2014.

[135] Ethan Ethan Katsh, M Ethan Katsh, and Janet Rifkin. *Online dispute resolution: Resolving conflicts in cyberspace*. John Wiley & Sons, Inc., 2001.

[136] David G Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[137] Özgür KISI and Erdal Uncuoglu. Comparison of three back-propagation training algorithms for two case studies. *Indian journal of engineering & materials sciences*, 12(5):434–442, 2005.

[138] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[139] Michael Kuperberg, Nikolas Herbst, Joakim von Kistowski, and Ralf Reussner. Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms. Technical report, KIT - University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association, 2011.

[140] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 2014.

[141] Hyperledger Sawtooth Lake. https://github.com/hyperledger/sawtooth-core/.

[142] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[143] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[144] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[145] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[146] Hazelcast. The leading in-memory Data Grid. `https://hazelcast.com/`.

[147] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2), 2010.

[148] SUNFISH Service Ledger. `https://github.com/sunfish-prj/Service-Ledger`.

[149] Hyo-Jin Lee, Myung-Sup Kim, James W Hong, and Gil-Haeng Lee. Qos parameters to network performance metrics mapping for sla monitoring. *KNOM Review*, 5(2):42–53, 2002.

[150] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[151] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 500(2011):292, 2011.

[152] Tsung. Distributed load testing tool. `http://tsung.erlang-projects.org/`.

[153] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):572–585, 2018.

[154] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[155] L Lundy and R Pradeep. On the migration from enterprise management to integrated service level management. *IEEE network*, 16(1):8–14, 2002.

[156] Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. *IACR Cryptology ePrint Archive*, 2015:1168, 2015.

[157] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *CCS*, pages 17–30. ACM, 2016.

[158] Vikram Makhija, Bruce Herndon, Paula Smith, Lisa Roderick, Eric Zamost, and Jennifer Anderson. Vmmark: A scalable benchmark for virtualized systems. 2006.

[159] Nathan Marz. Twitter Rolling Top-K Words Storm Topology. https://storm.apache.org/javadoc/apidocs/storm/starter/.

[160] J. Mattila. The blockchain phenomenon. *Berkeley Roundtable of the International Economy*, 2016.

[161] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting slas in clouds using rules. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 455–466, Berlin, Heidelberg, 2011. Springer-Verlag.

[162] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. BigchainDB: A Scalable Blockchain Database (DRAFT). 2016.

[163] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[164] Daniel A Menascé. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, 2002.

[165] John F Meyer and William H Sanders. Specification and construction of performability models. In *Proceedings of the Second International Workshop on Performability Modeling of Computer and Communication Systems*, pages 28–30, 1993.

[166] Haibo Mi, Huaimin Wang, Gang Yin, Yangfan Zhou, Dianxi Shi, and Lin Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521, 2010.

[167] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun. A review on consensus algorithm of blockchain.

[168] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press, Cambridge, expanded edition*, 19:88, 1969.

[169] MongoDB. http://www.mongodb.com/.

[170] Laura R. Moore, Kathryn Bean, and Tariq Ellahi. Transforming reactive auto-scaling into proactive auto-scaling. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 7–12, New York, NY, USA, 2013. ACM.

[171] David Mosberger and Tai Jin. httperf: a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[172] Multichain. https://www.multichain.com/.

[173] In Jae Myung. Tutorial on Maximum Likelihood Estimation. *Journal of Mathematical Psychology*, 47(1):90–100, February 2003.

[174] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available at `https://bitcoin.org/bitcoin.pdf`.

[175] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 137–148. IEEE, 2015.

[176] Neo4j. http://neo4j.com/.

[177] Docker Overlay Network. `https://docs.docker.com/network/overlay/`.

[178] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, 2013.

[179] Luca Nicoletti, Andrea Margheri, Federico Lombardi, Vladimiro Sassone, and Francesco Paolo Schiavo. Cross-cloud management of sensitive data via blockchain: a payslip calculation use case. In *ITA-SEC*, volume 2058. CEUR-WS.org, 2018.

[180] Abid Nisar, Waheed Iqbal, Fawaz S Bokhari, and Faisal Bukhari. Hybrid auto-scaling of multi-tier web applications: A case of using amazon public cloud.

[181] NodeJS. `https://nodejs.org`.

[182] Proof of Authority. https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains.

[183] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[184] OpenNebula. `http://opennebula.org/`.

[185] OpenStack. `https://www.openstack.org/`.

[186] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM.

[187] Jonathan W Palmer, Joseph P Bailey, and Samer Faraj. The role of intermediaries in the development of trust on the www: The use and prominence of trusted third parties and privacy statements. *Journal of Computer-Mediated Communication*, 5(3), 2000.

[188] Sang-Min Park and Marty Humphrey. Self-tuning virtual machines for predictable escience. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 356–363, Washington, DC, USA, 2009. IEEE Computer Society.

[189] Serguei Popov. A probabilistic analysis of the nxt forging algorithm. *Ledger*, 1:69–83, 2016.

[190] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[191] Rackspace. `https://www.rackspace.com/`.

[192] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: A reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 137–146, New York, NY, USA, 2009. ACM.

[193] SUNFISH Repository. `https://github.com/sunfish-prj/`.

[194] Vito Ricci. Fitting Distributions with R. *Contributed Documentation available on CRAN*, 2005.

[195] David Richey, Marie King, Lisa Bernhard, and David Van Horn. Method and system for facilitating electronic dispute resolution, April 8 2008. US Patent 7,356,516.

[196] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks*, pages 586–591. IEEE, 1993.

[197] Nicolo Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, and Bruno Sericola. Online scheduling for shuffle grouping in distributed stream processing systems. In *Proceedings of the ACM/IFIP/USENIX Middleware Conference*, 2016.

[198] Nicolo Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*, June 2015.

[199] Frank Rosenblatt. A model for experiential storage in neural networks. *Computer and information sciences. Washington, DC: Spartan*, 1964.

[200] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, 2011.

[201] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[202] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[203] Thiago Teixeira Sá, Rodrigo N Calheiros, and Danielo G Gomes. Cloudreports: An extensible simulation tool for energy-aware cloud computing environments. In *Cloud Computing*, pages 127–142. Springer, 2014.

[204] L. S. Sankar, M. Sindhu, and M. Sethumadhavan. Survey of consensus protocols on blockchain applications. In *ICACCS*, pages 1–5. IEEE, 2017.

[205] Francesco Paolo Schiavo, Vladimiro Sassone, Luca Nicoletti, and Andrea Margheri. FaaS: Federation-as-a-Service, 2016. Technical Report. Available at `https://arxiv.org/abs/1612.03937`.

[206] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[207] Scott Schneider, Martin Hirzel, and Buğra Gedik. Tutorial: stream processing optimizations. In *DEBS '13*, pages 249–258. ACM, 2013.

[208] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[209] Hanan Shteingart, Tal Neiman, and Yonatan Loewenstein. The role of first impression in operant learning. *Journal of Experimental Psychology: General*, 142(2):476, 2013.

[210] Rubis. Auction site prototype. `http://rubis.ow2.org/`.

[211] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[212] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[213] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013(881), 2013.

[214] Mehdi Sookhak, Abdullah Gani, Hamid Talebian, Adnan Akhunzada, Samee U. Khan, Rajkumar Buyya, and Albert Y. Zomaya. Remote Data Auditing in Cloud Computing Environments: A Survey, Taxonomy, and Open Issues. *ACM Comput. Surv.*, 47(4):65:1–65:34, May 2015.

[215] Roberto Baldoni Federico Lombardi Andrea Margheri Stefano De Angelis, Leonardo Aniello and Vladimiro Sassone. Pbft vs proof-of-authority:applying the cap theorem to permissioned blockchain. In *ITA-SEC*, volume 2058. CEUR-WS.org, 2018.

[216] Apache Storm. `http://storm.apache.org`.

[217] Bojan Suzic, Bernd Prünster, Dominik Ziegler, Alexander Marsalek, and Andreas Reiter. Balancing Utility and Security: Securing Cloud Federations of Public Entities. In *C&TC*, volume 10033 of *LNCS*, pages 943–961. Springer, 2016.

[218] Swagger. `https://swagger.io/`.

[219] Tim Swanson. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. *Report, available online, Apr*, 2015.

[220] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society.

[221] Bitcoin Wiki: Irreversible Transactions. https://en.bitcoin.it/wiki/Irreversible_Transactions/.

[222] L. Tseng. Recent results on fault-tolerant consensus in message-passing networks. In *Int. Colloquium on Structural Information and Communication Complexity*, pages 92–108. Springer, 2016.

[223] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1–1:39, March 2008.

[224] Lalitha Vaidyanathan, John Quinn, Ahmed Khaishgi, and Cara Cherry-Lisco. System and method for resolving a dispute in electronic commerce and managing an online dispute resolution process, August 5 2003. US Patent App. 10/634,654.

[225] Arjen Van Ooyen and Bernard Nienhuis. Improving the convergence of the back-propagation algorithm. *Neural Networks*, 5(3):465–471, 1992.

[226] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Quality of Service, 2004. IWQOS 2004. Twelfth IEEE International Workshop on*, pages 57–66, 2004.

[227] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[228] Thomas P Vogl, JK Mangis, AK Rigler, WT Zink, and DL Alkon. Accelerating the convergence of the back-propagation method. *Biological cybernetics*, 59(4-5):257–263, 1988.

[229] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[230] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[231] Michael K Weir. A method for self-determination of adaptive learning rates in back propagation. *Neural Networks*, 4(3):371–379, 1991.

[232] Mor Weiss, Boris Rozenberg, and Muhammad Barham. Practical Solutions For Format-Preserving Encryption. *CoRR*, abs/1506.04113, 2015.

[233] Bill Wilder. *Cloud architecture patterns: using microsoft azure*. " O'Reilly Media, Inc.", 2012.

[234] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[235] Hua Xiao, Brian Chan, Ying Zou, Jay W Benayon, Bill O'Farrell, Elena Litani, and Jen Hawkins. A framework for verifying sla compliance in composed services. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 457–464. IEEE, 2008.

[236] Jing Xu, Ming Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, pages 25–25, 2007.

[237] Jinhui Yao, Shiping Chen, Chen Wang, David Levy, and John Zic. Accountability as a service for the cloud. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 81–88. IEEE, 2010.

[238] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.

[239] Guoqiang Zhang, B. Eddy Patuwo, and Michael Y. Hu. Forecasting With Artificial Neural Networks: the State of the Art. *International Journal of Forecasting*, 14(1):35 – 62, 1998.

[240] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 27–, Washington, DC, USA, 2007. IEEE Computer Society.