Linear Scalability of Distributed Applications

THÈSE N^{O} 5278 (2012)

PRÉSENTÉE LE 3 FÉVRIER 2012

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'INFORMATION RÉPARTIS
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nicolas Bonvin

acceptée sur proposition du jury:

Prof. B. Faltings, président du jury
Prof. K. Aberer, directeur de thèse
Prof. A. Ailamaki, rapporteur
Prof. D. Kossmann, rapporteur
Prof. C. Pu, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2012

Résumé

L'explosion des applications sociales, du commerce électronique ou de la recherche sur Internet a créé le besoin de nouvelles technologies et de systèmes adaptés au traitement et à la gestion efficace d'un nombre considérable de données et d'utilisateurs. Ces applications doivent fonctionner sans interruption tous les jours de l'année, et doivent donc être capables de survivre à de subites et brutales montées en charge, ainsi qu'à toutes sortes de défaillances logicielles, matérielles, humaines et organisationnelles.

L'augmentation (ou la diminution) élastique et extensible des ressources affectées à une application distribuée, tout en satisfaisant des exigences de disponibilité et de performance, est essentielle pour la viabilité commerciale mais présente de grands défis pour les infrastructures actuelles. En effet, le *Cloud Computing* permet de fournir des ressources à la demande: il devient désormais aisé de démarrer des dizaines de serveurs en parallèle (ressources de calcul) ou de stocker un très grand nombre de données (ressources de stockage), même pour une durée très limitée, en payant uniquement pour les ressources consommées. Toutefois, ces infrastructures complexes constituées de ressources hétérogènes à faible coût sont sujettes aux pannes. En outre, bien que les ressources offertes par le *Cloud Computing* soient considérées comme pratiquement illimitées, seule une gestion adéquate de celles-ci peut répondre aux exigences des clients et ainsi éviter de trop fortes dégradations de performance.

Dans cette thèse, nous nous occupons de la gestion adaptative de ces ressources conformément aux exigences spécifiques des applications. Tout d'abord, nous abordons le problème de la gestion des ressources de stockage avec garanties de disponibilité dans un environnement intra-*cloud* et trouvons leur allocation optimale de manière décentralisée grâce à une économie virtuelle. Notre approche répond efficacement aux augmentations soudaines de charge ou aux pannes et profite de la distance géographique entre les nœuds pour améliorer la disponibilité des données.

Nous proposons ensuite une approche décentralisée de gestion adaptative des ressources de calcul pour des applications nécessitant une haute disponibilité et des garanties de performance lors de pics de charge, de pannes soudaines ou de mises à jour des ressources. Notre approche repose sur une économie virtuelle entre les différents composants de l'application et sur un système propageant en cascade les objectifs de performance des composants individuels de manière à satisfaire les exigences globales de l'application. Notre approche parvient à répondre aux exigences de l'application avec le minimum de ressources.

Enfin, comme les vendeurs de stockage proposent des tarifs pouvant varier considérablement, nous présentons une méthode inter-*cloud* d'allocation de ressources de stockage provenant de plusieurs vendeurs en fournissant à l'utilisateur un système qui garantit le meilleur tarif pour stocker et servir ses données, tout en satisfaisant ses besoins en matière de disponibilité, durabilité, latence, etc. Notre système optimise en permanence le placement des données en fonction de leur type et de leur mode d'utilisation, tout en minimisant les coûts de migration d'un vendeur à un autre, afin d'éviter de rester prisonnier de l'un d'eux.

Mots Clés: base de données, architecture, modèle économique, cloud computing, haute-disponibilité, extensibilité

Abstract

The explosion of social applications such as Facebook, LinkedIn and Twitter, of electronic commerce with companies like Amazon.com and Ebay.com, and of Internet search has created the need for new technologies and appropriate systems to manage effectively a considerable amount of data and users. These applications must run continuously every day of the year and must be capable of surviving sudden and abrupt load increases as well as all kinds of software, hardware, human and organizational failures.

Increasing (or decreasing) the allocated resources of a distributed application in an elastic and scalable manner, while satisfying requirements on availability and performance in a cost-effective way, is essential for the commercial viability but it poses great challenges in today's infrastructures. Indeed, *Cloud Computing* can provide resources on demand: it now becomes easy to start dozens of servers in parallel (computational resources) or to store a huge amount of data (storage resources), even for a very limited period, paying only for the resources consumed. However, these complex infrastructures consisting of heterogeneous and low-cost resources are failure-prone. Also, although cloud resources are deemed to be virtually unlimited, only adequate resource management and demand multiplexing can meet customer requirements and avoid performance deteriorations.

In this thesis, we deal with adaptive management of cloud resources under specific application requirements. First, in the intra-cloud environment, we address the problem of cloud storage resource management with availability guarantees and find the optimal resource allocation in a decentralized way by means of a virtual economy. Data replicas migrate, replicate or delete themselves according to their economic fitness. Our approach responds effectively to sudden load increases or failures and makes best use of the geographical distance between nodes to improve application-specific data availability.

We then propose a decentralized approach for adaptive management of computational resources for applications requiring high availability and performance guarantees under load spikes, sudden failures or cloud resource updates. Our approach involves a virtual economy among service components (similar to the one among data replicas) and an innovative cascading scheme for setting up the performance goals of individual components so as to meet the overall application requirements. Our approach manages to meet application requirements with the minimum resources, by allocating new ones or releasing redundant ones.

Finally, as cloud storage vendors offer online services at different rates, which can vary widely due to second-degree price discrimination, we present an inter-cloud storage resource allocation method to aggregate resources from different storage vendors and provide to the user a system which guarantees the best rate to host and serve its data, while satisfying the user requirements on availability, durability, latency, etc. Our system continuously optimizes the placement of data according to its type and usage pattern, and minimizes migration costs from one provider to another, thereby avoiding vendor lock-in.

Keywords: database, architecture, economic model, cloud computing, high-availability, scalability

Acknowledgements

Let me begin by thanking my thesis supervisor, Prof. Dr. Karl Aberer, who helped me throughout my Ph.D and allowed me to work on topics that interested me particularly. I also thank the members of my thesis committee for their constructive and encouraging comments: Prof. Dr. Anastasia Ailamaki, Prof. Dr. Boi Faltings, Prof. Dr. Donald Kossmann and Prof. Dr. Calton Pu.

A big thank you to my officemates, to Chantal and to all members of the LSIR lab. I'm particularly thankful to Thanasis for his invaluable assistance, work and friendship throughout my Ph.D, and Ho Young for his support and friendship.

I also want to thank my parents and my family for their unfailing support, my flatmates, my friends and all who are dear to me and supported me.

Finally, I would like to deeply thank Nicolas Cordonier who helped me make the right choice at an important moment, Claire-Lise who gave me confidence in myself, and Rachel who encouraged me throughout this wonderful adventure.

Thank you.

Contents

Contents **v**

List of Figures **ix**

List of Tables **xiii**

List of Algorithms **xv**

I Introduction **1**

1 Introduction **3**
1.1 Distributed Applications 5
1.1.1 Service-Orientation 6
1.1.2 Cloud Computing 7
1.2 Scalability 8
1.2.1 A Motivating Example 10
1.3 Contribution of the Work 13
1.4 Scope and Limitations 15
1.5 Structure of the Thesis 16
1.6 Selected Publications 16

II State of the Art **19**

2 Distributed Application Scalability **21**
2.1 Introduction 21
2.2 Scalability Best Practises 24
2.3 Cloud Computing 26
2.3.1 Definition 26
2.3.2 Benefits 29
2.3.3 Cloud Storage 31
2.4 Conclusion 33

3 Database Scalability **35**

3.1 Introduction 35
3.2 Relational Databases 37
3.2.1 Replication 37
3.2.2 Database Sharding 39
3.3 NoSQL Databases 44
3.4 NewSQL Databases 47
3.5 Consistency Models 48
3.5.1 Strong Consistency 48
3.5.2 Eventual Consistency 51
3.6 Conclusion 52

III Contributions 55

4 Building Highly-Available and Scalable Cloud Storage 57
4.1 Introduction 57
4.2 *Skute*: Scattered Key-Value Store 59
4.2.1 Physical Node 59
4.2.2 Virtual Node 60
4.2.3 Virtual Ring 60
4.2.4 Routing 62
4.2.5 Data Consistency 62
4.3 Problem Definition 63
4.3.1 Maximize Data Availability 63
4.3.2 Minimize Communication Cost 64
4.3.3 Maximize Net Benefit 64
4.4 The Individual Optimization 65
4.4.1 Board 65
4.4.2 Physical Node 66
4.4.3 Maintaining Availability 66
4.4.4 Virtual Node Decision Tree 68
4.5 Equilibrium Analysis 71
4.6 Rational Strategies 73
4.7 Simulation Results 74
4.7.1 The Simulation Model 74
4.7.2 Convergence to Equilibrium and Optimal Solution . . . 75
4.7.3 Fault Tolerance against Correlated Failures and Adaptation to New Resources 75
4.7.4 Adaptation to the Query Load 78
4.7.5 Scalability of the Approach 81
4.8 Implementation and Experimental Results in a Real Testbed . . 83
4.8.1 Verification of Simulation Results 84
4.8.2 Scalable Performance 84
4.8.3 Adaptivity to Varying Load 86
4.8.4 Adaptivity to Failure 86
4.9 Potential Applications 87
4.10 Related Work 88
4.11 Conclusion 90

5 Building Highly-Available and Scalable Cloud Applications 91

5.1 Introduction 92
5.2 Motivation 94
5.2.1 Running Example 94
5.3 *Scarce*: the Quest of Autonomic Applications 96
5.3.1 The Approach 96
5.3.2 Server Agent 96
5.3.3 Routing Table 97
5.3.4 Economic Model 98
5.4 Maintaining High-Availability 101
5.5 Meeting SLA Performance Guarantees 102
5.5.1 Cascading Performance Constraints 102
5.6 Automatic Provisioning of Cloud Resources 104
5.6.1 Adaptivity to Slow Servers 104
5.7 Evaluation 105
5.7.1 Scalability, High-Availability and Load-Balancing 106
5.7.2 SLA Performance Guarantees 111
5.8 Related Work 118
5.9 Conclusion 119

6 Federation of Cloud Storage **121**
6.1 Introduction 121
6.2 Motivation 123
6.2.1 Avoiding Vendor Lock-in 123
6.2.2 Paying a Fair Price 124
6.3 Scalia: Multi-Cloud Storage 125
6.3.1 Engine Layer 126
6.3.2 Caching Layer 132
6.3.3 Database Layer 134
6.3.4 Life cycle of read and write operations 135
6.3.5 Private Storage Resources 137
6.3.6 Discussion 138
6.4 Evaluation 139
6.4.1 Experimental Setup 139
6.4.2 Slashdot Effect Scenario 140
6.4.3 Gallery Scenario 141
6.4.4 Adding Storage Resources 142
6.4.5 Active repair 146
6.4.6 Pricing Update 148
6.5 Related Work 148
6.6 Conclusions 149

IV Conclusion **151**

7 Conclusion **153**
7.1 Summary of the Work 153
7.2 Future Work 154
7.2.1 Improving the Current Work 154
7.2.2 Future Directions 155

Bibliography **157**

Curriculum Vitae **169**

List of Figures

1.1 Distributed computing vs parallel computing 5
1.2 Vertical vs horizontal Scalability 9
1.3 Scaling from 1 to 4 servers 10
1.4 Master-slave database replication 11
1.5 Master-slave replication does not scale 12
1.6 Master-slave replication limited by write operations 12

2.1 Stateful function 21
2.2 Stateless function 22
2.3 Storing sessions locally 22
2.4 Storing sessions centrally 23
2.5 Storing sessions at client side 24
2.6 Management responsibilities in cloud stack 28
2.7 Effort and opportunity in the cloud stack 29
2.8 Fixed vs on-demand capacity 30
2.9 Cloud Storage Gateways 32
2.10 Content Delivery Network 33

3.1 Master-slave replication 37
3.2 Tree replication 38
3.3 Multi-master replication 39
3.4 Database partitioning: horizontal vs vertical 40
3.5 Simplified data modelof a weblog 41
3.6 Vertical database partitioning 41
3.7 Horizontal database partitioning 42
3.8 Database sharding 43
3.9 NoSQL databases landscape 45
3.10 CAP theorem 46
3.11 Eventual Consistency 49
3.12 Strong Consistency: read operation 50
3.13 Strong Consistency: write failure 51

4.1 Ring topology 60
4.2 3 applications with different availability levels 61
4.3 Data consistency during node migration 63

4.4 Decision tree of the virtual node agent 70
4.5 Replication process at startup 76
4.6 Concurrent rack failures 77
4.7 *Skute*: robustness against upgrades and failures 77
4.8 Number of Queries per Partition 78
4.9 Number of Virtual Nodes per Partition 78
4.10 Total amount of virtual nodes with uniform load 79
4.11 Average query load per virtual ring with evenly distributed queries 80
4.12 Average query load per server with evenly distributed queries . . 80
4.13 Average virtual rent price with evenly distributed queries 81
4.14 Average query load per virtual ring with unevenly distributed queries 82
4.15 Average query load per server with unevenly distributed queries . 82
4.16 Storage saturation 83
4.17 Network saturation 84
4.18 Average query load with unevenly distributed queries 85
4.19 Response time and throughput 85
4.20 Load peak 86
4.21 Server crash 87
4.22 Using *Skute* as a cache 88
4.23 Using *Skute* in an anycast DNS service 89

5.1 An application on a cloud computing infrastructure 95
5.2 Example of a distributed application 95
5.3 Server and agents 97
5.4 Contention level of servers 99
5.5 SLA propagation 103
5.6 Statistics about the response time 105
5.7 Response time of different placement approaches 108
5.8 Throughput of different placement approaches 108
5.9 Framework scalability 109
5.10 Failure percentages during crashes 109
5.11 Adding new cloud resources 110
5.12 Routing policies 110
5.13 Architecture of a test application 111
5.14 Resources consumed 112
5.15 Mean response time 112
5.16 95th percentile response time 113
5.17 Throughput when load varies 113
5.18 Computed SLA constraints 114
5.19 SLA violations 115
5.20 Response time in case of slow servers 115
5.21 Resources used in case of slow servers 116
5.22 Response times during scalability experiment 117
5.23 Throughput during scalability experiment 117
5.24 Resources used during scalability experiment 118

6.1 Erasure coding 123
6.2 Multi datacenter architecture 126
6.3 Time left to live for a class of objects 128

6.4 Classification of objects 128
6.5 Periodic Optimization 131
6.6 Trend detection (1 day decision period) 133
6.7 Trend detection (1 week decision period) 134
6.8 Concurrent writes 136
6.9 Amount of resources for the Slashdot scenario 139
6.10 Total cost of the Slashdot scenario 140
6.11 Amount of resources for the gallery scenario 141
6.12 Total cost for the gallery scenario 141
6.13 Resources used when adding a public storage provider 143
6.14 Total cost when adding a public storage provider 143
6.15 Amount of resources when adding a private storage resource . . . 144
6.16 Cost per hour when adding a private storage resource 144
6.17 Cumulative cost when adding a private storage resource 145
6.18 Total cost when adding a private storage resource 145
6.19 Active repair 147
6.20 Pricing update 147

List of Tables

1.1 Availability measurement 5

4.1 Example of quorum parameters 63
4.2 Parameters of small-scale and large-scale experiments. 76

5.1 The local routing table 97

6.1 Example of storage rules 124
6.2 Example of provider sets 124
6.3 Providers' abbreviations 124
6.4 Example of providers prices 125
6.5 Metadata of a file 137
6.6 Sets of providers 139

List of Algorithms

5.1 Updating the replicas' coefficients 106
6.1 Computing the best set of providers 129
6.2 Computing the best threshold of a set of providers 130
6.3 Trend detection: *alert()* function 132
6.4 Trend detection 133

Part I

Introduction

Chapter 1

Introduction

Communicating is the essence of human beings. In recent years, electronic means, such as email or mobile phone, have changed the way people interact. A large proportion of individuals and a majority of businesses communicate using digital networks, usually connected to Internet. With the democratization of Internet access, virtual social networks have emerged in the digital world, reproducing and even amplifying the social interactions occurring in the real world. The dissemination of information to a wide spectrum of different people becomes near real time, thanks to communication tools such as Facebook [13] or Twitter [50] which offer greater reactivity compared to traditional media like radio or television. The way of consuming is also changing, reflecting the impressive growth of electronic commerce, as demonstrated by the success of e-commerce companies such as Amazon.com [3]. Such platforms allowing a huge number of people to communicate instantaneously, dealing with a considerable number of users or customers, or managing a vast amount of data, require a very high degree of reliability and should tackle every kind of failure gracefully in order to remain trusted and useful. As a consequence of these new communication tools, popular events are relayed quickly to an impressive number of people and can cause virtual flash crowds, which can easily take down computer infrastructures. Imagine a company having the chance to run a website that becomes popular, a large number of Internet users might suddenly want to access the site at the same time. Such a company should be well prepared for this situation in order to cope with the sudden load and remain successful. To build a robust web application, a very important aspect is to be able to grow and shrink the infrastructure rapidly and at a limited cost. An application architecture for a highly demanded website needs to achieve three goals:

1. **scalability**, which is characterized by the ability to gracefully handle additional traffic or data growth while maintaining service quality and maintainability of a computer system;
2. **high-availability**, which refers to a computer system that is continuously operational for a contractual measurement period;
3. **performance**, which corresponds to the amount of useful work accomplished by a computer system compared to the time and resources used.

To be flexible enough to support the addition or removal of resources, an application has to be split into several components so that they can be distributed among the available resources. The design of a distributed application involves many trade-offs that need to be considered when building a solution. Should an application be very fast for a limited amount of users (focusing on performance), or should the application deliver acceptable performance for a large amount of users (focusing on scalability), or maybe both? The concepts of scalability and performance are often linked, but they are distinct: performance measures the time taken for a request to be executed, whereas scalability measures the ability of the system to maintain the performance of a request under growing load.

Similarly, should an application be designed to maximize throughput or minimize latency? A good trade-off would probably be to strive for maximal throughput with acceptable latency. A distributed application will also have to deal with consistency, availability and network partitions tolerance, but as described in Section 3.3, achieving all three properties at the same time is not possible; therefore several architectural decisions have to be taken at design time by choosing the objectives the application should be optimized for.

In distributed applications, besides scalability, there are others reasons for adding additional resources to the system: redundancy is mandatory to make a system highly-available. An application and its underlying infrastructure should be designed to ensure business continuity and to avoid unplanned downtime. A properly designed system should handle transparently hardware and software failures without affecting the ability of an end user to access the application. Within a local site, an highly-available application is usually deployed across several identical servers and can operate basically in two modes:

- **active-passive**: passive nodes are in a standby configuration and do not serve requests until a failure is detected in the software or hardware of one of the active nodes; in that case, a passive node becomes active, replacing the failed node. The number of passive nodes is a tradeoff between cost and reliability requirements. In a two nodes setup, this mode is usually called *failover*.
- **active-active**: all nodes serve requests. When one of the nodes has a software or hardware failure, the surviving nodes take over the application load of the failed node and requests are served by the remaining active nodes. This mode eliminates the need for standby nodes, but requires extra capacity on all active nodes to cope with the load of failing nodes.

The traditional metric to evaluate the availability of a system is the uptime, which corresponds to the fraction of time the application is handling requests. It is typically measured in *nines* as shown in Table 1.1 and can be defined as:

$$availability = MTBF/(MTBF + MTTR) \tag{1.1}$$

where $MTBF$ is a metric representing the *meantime-between-failure* and $MTTR$ corresponds to the *mean-time-to-repair*. Given Eq. 1.1, the uptime of an application can be improved either by reducing the time to fix failures or by reducing their frequency.

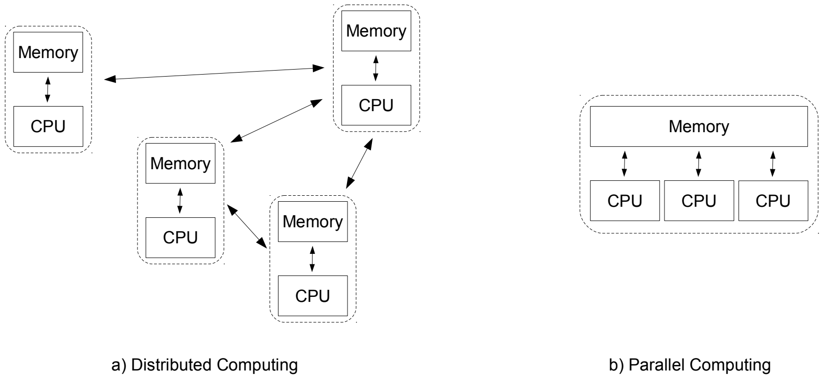
Table 1.1: Availability measurement

Availability %	*Downtime per year*	*Downtime per month*
90% (“one nine”)	36.5 days	72 hours
95%	18.25 days	36 hours
98%	7.30 days	14.4 hours
99% (“two nines”)	3.65 days	7.20 hours
99.5%	1.83 days	3.60 hours
99.8%	17.52 hours	86.23 minutes
99.9% (“three nines”)	8.76 hours	43.2 minutes
99.95%	4.38 hours	21.56 minutes
99.99% (“four nines”)	52.56 minutes	4.32 minutes
99.999% (“five nines”)	5.26 minutes	25.9 seconds
99.9999% (“six nines”)	31.5 seconds	2.59 seconds

However, adding more resources and redundancy does not automatically result in a more performant, more scalable or more available system: the application has to be developed carefully from the beginning, and should be deployed among distinct geographical zones. Indeed, a scalable application with a very high level of redundancy may still be unreachable if it has been deployed in a single site: the datacenter itself should be seen as a single point of failure.

Designing, building and deploying an application that is at the same time performant, scalable and highly-available is complex and presents a wide range of architectural, technical and operational challenges.

1.1 Distributed Applications



a) Distributed Computing

b) Parallel Computing

Figure 1.1: Distributed computing versus parallel computing

Historically, a distributed system has been considered as a network of individual computers, or *nodes*, which are physically distributed within several geographical areas. However, the concept has evolved and now it can also describe systems with autonomous processes running on the same physical node. A distributed system comprises autonomous processes, or *components*, which have their own dedicated local memory and communicate by message passing. In parallel computing, a shared memory is accessed and used to exchange data among all processors, contrary to distributed computing where each processor has its own memory, as depicted in Figure 1.1. That is, a distributed application comprises distinct components physically located on various computer systems and interacting by exchanging messages over a network.

In the following subsections, we discuss the key technologies underlying large-scale distributed applications, namely *service-orientation*, which is a design paradigm to build applications in the form of services and *cloud computing*, which allows to use on-demand computing and storage resources.

1.1.1 Service-Orientation

A *service-oriented architecture (SOA)* is governed by a flexible set design principles ensuring the separation of concerns in the application. Following this architecture, the application is split into a set of interoperable services, which are distributed software components partitioned into operational capabilities, each designed to address an individual concern. A service is characterized [84] by eight key aspects:

- **Standardized service contract**: all services of the same technical system are exposed through contracts following the same rules of standardization.
- **Loose coupling**: coupling refers to the degree of direct knowledge that one service has of another. In a loosely coupled system, the interdependencies between the services are reduced to the minimum. The contract of a service must therefore impose a loose coupling of its clients. It is essential to avoid the technical and functional coupling between consumers and service providers: the coupling requires the consumer to know the exchange protocol and format of the provider, complicating or even prohibiting the development of the application on a larger scale. The consumers of a service should be only linked to the service contract and not to the service itself.
- **Abstraction**: the service contract should only contain essential information to its invocation. Only such information should be published. The principle of abstraction is to provide the services as black boxes. The only available information to the consumers of a service are those contained in its contract. Thus, the designers and developers of a software component consuming a service are not aware of the implementation of the service.
- **Reusability**: a service expresses a logic independent of any particular business process or technology and can be therefore positioned as a reusable resource. The implementation of an application following a

service-oriented architecture is intended, among other things, to avoid wasting resources by eliminating redundancies.

- **Autonomy**: a service must exercise strong control over its underlying execution environment. A service must comply with its contract, which usually includes a service-level agreement (SLA [1]), regardless of the volume of requests it has to process. Concurrent accesses to a service should not change in any way its behavior, reliability or performance. The stronger the control, the more predictable the service execution.
- **Statelessness**: A service must minimize the resource consumption by delegating the management of state information when necessary. The responsibility for state management is delegated to the service consumers. Delegating the state management to the clients meets the REST [86] design principles of web services.
- **Discoverability**: a service is complemented by a set of communication metadata through which it can be discovered and interpreted effectively.
- **Composability**: a service must be designed to participate in service compositions. The objective is to determine the ”right” granularity of services in order to decompose the solution to a high-level business problem into a set of ”smaller reusable processing units” (i.e., the services). The idea is to be able to reconstruct indefinitely the business logic inside high-level composite services.

While building an application following the aforementioned principles is a good start, several aspects have to be considered before having a highly-available and scalable application. An application composed of many services is useful only when all services are available and able to perform their respective tasks. If a single service fails or is overloaded, then the whole application is impacted. Therefore, services have to be replicated to achieve high-availability and to be able to cope with growing load. Replicating a service should first take into account the geographical location of the other replicas, such that two replicas of the same service are not physically hosted on the same rack or the same datacenter, so as to avoid correlated failures. Second, the replication process should also take into account the available resources (CPU, memory, disk space, network) of the servers dedicated to the application in order to choose the best suited server for hosting a new replica of the service. These replication problems are addressed in the context of distributed databases where data partitions are cleverly replicated and migrated (see Chapter 4), as well as in the context of distributed applications composed of many services (see Chapter 5).

1.1.2 Cloud Computing

Since a couple of years, it is possible to rent dedicated servers in well connected datacenters and more recently, virtualization solutions have enabled a new way of provisioning on-demand computing and storage resources: *cloud*

[1] `http://en.wikipedia.org/wiki/Service_level_agreement`

computing is “a model for enabling 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” [62].

Apart from the aspects of privacy and data security, which are among the main obstacles to the wide adoption of cloud computing, building an application that takes advantage of a cloud computing infrastructure presents several challenges. Unlike traditional distributed applications that rely on dedicated servers with predictable performance, an application developed for the cloud must adapt to the underlying virtualized, ephemeral and unreliable infrastructure, whose performance can vary suddenly. Moreover, modern distributed applications should derive advantage of the *elasticity* of cloud computing, which refers to the ability to amplify and instantly upgrade storage, computing and network resources on demand. They should also be able to dynamically react to changing conditions, such as software and hardware failures or quickly varying volume of requests, by dynamically adapting the location and the amount of replicas of the services composing the application. Lacking the complete control of the infrastructure, deploying, updating and monitoring cloud-based applications becomes more complex. Therefore, building autonomic and adaptive applications that only consume the minimum resources necessary for their tasks opens interesting research areas. In Chapter 5, we will present a framework addressing these problems specific to applications deployed on a cloud computing infrastructure.

1.2 Scalability

A very concise and generic definition of *scalability* was given in [143]:

> Essentially, it means how well a particular solution fits a problem as the scope of that problem increases.

In other words, scalability corresponds to the ability to gracefully handle additional traffic or data growth while maintaining service quality and maintainability of the system. A system whose capacity can be augmented by adding more of the same software or hardware is called *horizontally* scalable and this is the only true form of scalability as the system capacity can grow almost indefinitely. An architecture should allow the system capacity to be increased or decreased when the application needs change. The efficiency of an horizontally scalable system increases (or decreases) after hardware is added (or removed) proportionally to the capacity added (or removed). On the other hand, a system not able to scale horizontally (i.e., to scale out) can be scaled up by adding resources (processors, storage, and/or RAM) to the server running the application to achieve greater productivity: this is referred to as *vertical* scalability. Vertical scalability is bounded (e.g., one cannot add an arbitrarily large amount of CPUs or RAM), expensive and proprietary. In addition to its ability to grow to an almost infinite capacity, a system based on a horizontally scalable architecture proves less costly for a given capacity than a system based on a vertically scalable architecture, as depicted in Figure 1.2. A scalable system has the following characteristics:

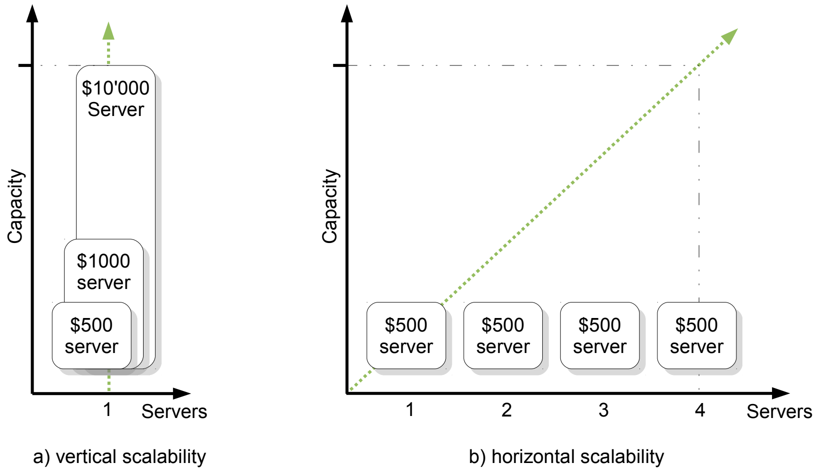


Figure 1.2: Vertical vs horizontal scalability: the price growth for vertical scaling is exponential. For a system with the same capacity, the vertical approach costs 10'000$ while the horizontal one costs only 2'000$

- it becomes more cost effective when the system grows, thanks to economies of scale;
- it is able to deal with semantic and resources heterogeneity;
- it is operationally efficient;
- it is resilient;
- increasing the allocated resources results in a proportional increase in term of capacity.

Scalability is an architectural concern related to maintainability of the system, traffic and dataset growth. However, both scalability and performance are important as an application should handle current commitments (performance) and also future ones (scalability). Performance and scalability have to act symbiotically in order to allow a cost-efficient growth of a business: a scalable application needs to have the capability to adapt to growing workload, respectively ensure a certain performance with a growing workload. For e-commerce websites, social networking sites and other large distributed applications, the ability to scale is directly related to the success or failure of the business. Several scalability requirements can be defined [138] for large applications:

- **user load scalability**: the application needs to cope with a fast growing number of users (potentially millions of users);
- **data load scalability**: the application needs to cope with a fast growing amount of data (potentially petabytes of data), which is either produced by a few users or by the aggregate of many users;
- **computational scalability**: operations on the data should be able to scale for both an increasing number of users and increasing data sizes;

- **scale adaptivity**: in order to cope with increasing or decreasing application load, the architecture and operational environment should provide the ability to add or remove resources quickly, without application changes or impact on the availability of the application.

Most of the scalable application architectures are based on function and data partitioning, sharing the work load on many servers. Functional partitioning aims at splitting an application into several loosely-coupled services each performing a specific task, following the service-oriented architecture approach. However, functional partitioning can be too limited to offer high scalability when the amount of users or data increases consequently and the load of a single functional area surpasses the capacity of a single machine. Therefore data partitioning should be used in conjunction with functional partitioning, such that the application work load is also spread over a set of data partitions hosted on different machines.

1.2.1 A Motivating Example

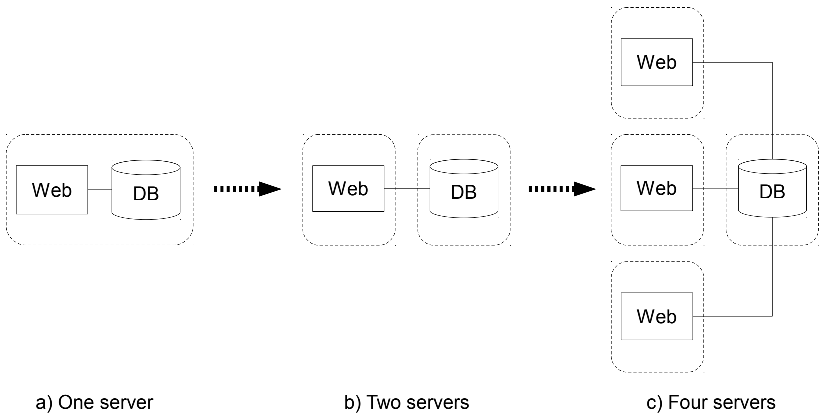


Figure 1.3: Scaling an application from one to four servers

Let us consider an example inspired by [87] of a simple web application composed of a web server and a database: what are the problems encountered when scaling the application from a single server to a few dozen servers?

The web server is responsible for handling the business logic and for preparing as well as sending the reply to the client. The database stores persistently the data needed by the application (i.e., the web site). In the simplest scenario, both the web server and the database can run on the same server. But eventually, the web site gets slow, and a point is reached where tuning the settings of the servers or of the application does not help anymore. Moreover, having only one server implies a high risk in case of failure, the server being a single point of failure (SPOF) as shown in Figure 1.3 a).

To improve the performance of the web site, the database is moved to its own dedicated server, as depicted in Figure 1.3 b). Now, instead of having a single

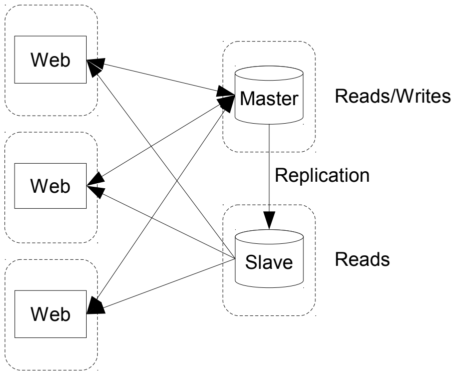


Figure 1.4: Master-slave database replication architecture.

point of failure, the architecture comprises two: there are no hot or cold spare servers to replace a failed server. As the web site starts to be successful, more and more people access the application, resulting again in poor performance. Now the web server reaches high CPU utilization as a result of growing traffic and is no more able to process requests with acceptable latency: more web servers are required. After having added two more web servers, the setup has now three servers dedicated to the application and a database server, as depicted in Figure 1.3 c). The users' requests are now balanced between the three web servers using a load-balancing mechanism. However, as a result of growing volume of requests sent by the web servers, the database reaches high I/O utilization and is no more able to process incoming requests with acceptable latency. Furthermore, the database is a single point of failure.

We discuss now how this issue is commonly resolved in this example situation and the reasons why database scaling is more complex. The most straightforward approach to improve the performance of the database layer is to add an additional database server and to configure the first one as master and the second one as slave, such that the write operations are performed on the master and the read requests can be handled by both the master and the slave, as depicted in Figure 1.4. To cope with growing load, additional web servers and database servers need to be added, hitting alternatively CPU bounds and I/O bounds. This architecture, as shown in Figure 1.5, is appropriate for moderate load, but performance will drop for high traffic conditions, as soon as the volume of write requests surpasses the capacity of one of the database servers: adding additional slave database servers will exacerbate the problems rather than solving them.

Besides the master being a single point of failure, this architecture does not improve the write scalability and only solves partially the read scalability issue. Moreover, maintaining such an infrastructure quickly becomes increasingly difficult. Although the read operations are spread among several servers, the scalability of reads is limited by the write operations that are replicated from the master to every slave. Let us imagine a server being able to handle 800 operations per second (either read or write). Consider the following load on a single server: 600 reads per second and 200 writes per

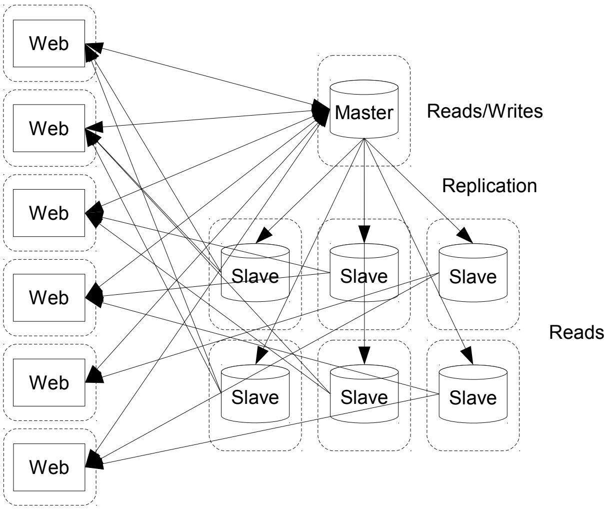


Figure 1.5: Master-slave database replication architecture is suitable only for moderated traffic web site: write operations are not scalable.

second (Figure 1.6 a)). If a second server (i.e., a slave) is added, the read operations will be shared between both servers while each server has still to perform all write operations, as depicted in Figure 1.6 b). When the amount of write operations increases, Figure 1.6 c) shows that the resources of the slaves will be eventually dedicated mostly to write operations, penalizing the performance of read operations. At this point, not only the web site is not able to handle the write load, but it also cannot serve the read requests with satisfactory performance.

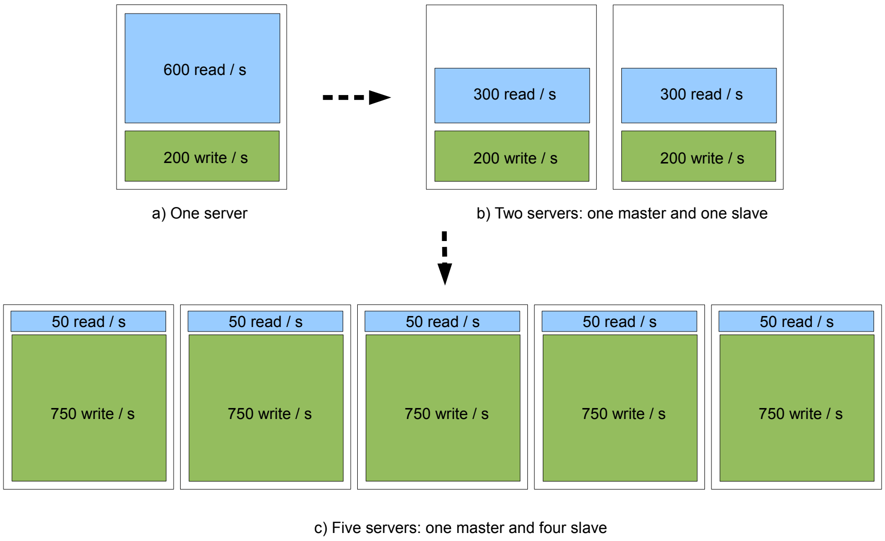


Figure 1.6: Master-slave database replication: the servers' resources will eventually be consumed by write operations.

In order to avoid that the write load consumes all the database server resources, one might think that adding additional master database servers would resolve the write scalability issue. Let us consider a write-mostly web application selling a finite amount of goods such as airline tickets, where the number of seats for a given flight is limited. Thus, the database has to enforce that only a fixed number of tickets, let us say 200, will be sold per flight. When a single master database is no more able to cope with the write load due to limited capacity, a second master database could be added, such that some write operations are performed on the first master database and the remaining write operations are processed by the second master database. If both master databases do not interact during a write operation, the web application might sell up to 400 tickets for a flight of only 200 seats, as each master database is allowed to sell up to 200 tickets. To avoid inconsistencies, both master databases have to cooperate in order to limit the number of sold tickets: a write operation has to be acknowledged by both master databases, thus removing the benefit in term of write scalability of having a second master database. As a write operation has to be performed by both master database servers, it has to succeed or fail as a unit, called a *transaction* [92], and therefore requires advanced techniques providing strong consistency in distributed transactions [93], such as 2 phase commit [60, 91, 131, 120], 3 phase commit [146], Paxos [114] or various approaches to state machine replication [70, 112]. As distributed transactions come with their own price, namely performance penalty and complexity, these techniques are mainly used to improve the fault-tolerance of a master database by adding additional replicas rather than improving its performance or scalability.

In this example, we first have seen that scaling the read operations using a common master-slave architecture is only appropriate for moderated load. Second, using multiple master database servers does not improve the write scalability as the integrity of data in a distributed transaction has to be ensured by non scalable distributed protocols introducing complexity and overhead. In order to cope with high traffic web applications, a scalable database layer has to be built using more sophisticated approaches, as explained in Section 3.2.

1.3 Contribution of the Work

In this thesis, we make the following contributions to database and cloud application scalability as well as to storage federation. In the first instance, we improve the cost-efficiency and the scalability of key-value stores, which are usually used to implement cloud storage solutions. In a second step, we propose an approach that allows an application to take advantage of the elasticity provided by a cloud computing infrastructure, while at the same time addressing its major challenges, namely the unpredictable performance and the ephemeral nature of cloud resources. Finally, to further improve the scalability of cloud storage, we propose a solution for federating multiple cloud storage providers, where the location of a data is determined at creation time and readjusted periodically, based on the constraints and the optimization objectives defined by the data owner.

In the area of database scalability we propose a scattered key-value store (Chapter 4) which is designed to provide high and differentiated data availability statistical guarantees to multiple applications in a cost-efficient way in terms of rent price and query response times in a cloud environment. Distributed key-value stores being a widely employed service case of cloud storage, our approach combines the following innovative characteristics:

- it enables a computational economy for cloud storage resources;
- it provides differentiated availability statistical guarantees to different applications despite failures by geographical diversification of replicas;
- it applies a distributed economic model for the cost-efficient self-organization of data replicas in the cloud that is adaptive to adding new storage resources, to node failures and to client locations;
- it efficiently and fairly utilizes cloud resources by performing load balancing in the cloud adaptively to the query load.

In the area of application scalability we propose a cost-efficient approach (Chapter 5) for dynamic and geographically-diverse replication of components in a cloud computing infrastructure that effectively adapts to load variations and offers service availability guarantees. Our approach combines the following unique characteristics:

- adaptive adjustment of cloud resource allocation in order to statistically satisfy response time or availability SLA requirements;
- cost-effective resource allocation and component placement for minimizing the operational costs of the cloud application;
- detection and removal or replacement of stale cloud resources;
- component replication and migration for accommodating load variations and for supple load balancing;
- decentralized self-management of the cloud resources for the application;
- geographically-diverse placement of clone component instances.

In the area of storage federation we introduce a system (Chapter 6) that continuously adapts the placement of data among several storage providers subject to optimization objectives, such cost minimization. Our system combines the following unique and novel characteristics:

- adaptive data placement based on the real-time data access patterns, so as to minimize the price that the data owner has to pay to the cloud storage providers given a set of customer rules, e.g., availability, durability, etc. Other optimization goals for data placement are also conceivable, such as:
  - maintaining a certain monthly budget by relaxing some constraints, such as lock-in or availability;

  - minimizing query latency by promoting the most high-performing providers;
- compliance of rules set by customers for a data, such as data durability, data availability and level of vendor lock-in;
- orchestration of a non-static set of public cloud and corporate-owned private storage resources;
- a robust distributed architecture for its implementation that is able to handle a large number of objects stored, which are accessed by a large number of potential users.

1.4 Scope and Limitations

This thesis focuses essentially on how to build scalable distributed applications accessible through the Web. Specifically, we focus on applications that are facing a rapid increase in terms of traffic, users or data. It is necessary for such applications to absorb the extra load quickly without having to rewrite the application or to change the architecture. Ideally, only the addition of new physical resources should be sufficient to ensure the smooth operation of the application. To meet these objectives, the class of applications called "scalable" should follow certain best practices in terms of architectural design and development. In particular, an application should be split into components that avoid as much as possible to share states, to access the same data at the same time (i.e., data contention) or to take decisions requiring the participation of several components (e.g., distributed transactions). Indeed, these practices have the effect of greatly limiting the ability of the application to scale. For example, sharing a state between two or more components requires a shared storage medium, which introduces data contention, or synchronization mechanisms, which become ineffective with a large state or when the amount of components to be synchronized increases. Avoiding synchronization between components and data contention, or reducing as much as possible coupling (i.e., dependencies between components) and shared or mutable states are not only recommended to build distributed applications, but are also considered as general good programming practices. Similarly, the application components should process requests asynchronously and in a non-blocking manner (e.g., with the help of queues), so as to handle load peaks more gracefully, by delaying requests that could not be processed immediately instead of dropping them.

Therefore, following current scalability best practices (see Section 2.2), the application components are thus considered in this thesis to be stateless, horizontally split, loosely coupled and do not require distributed transactions or strong consistency most of the time. In modern web applications, most of the requests do not require transactional guarantees and can be satisfied using only eventual consistency. Typical applications include social web applications managing thousands of users along with the large amount of data produced and shared by the users, or applications whose workload can be easily parallelized (e.g., transcoding music files). However, most of current frameworks (e.g., .NET [33] or Java EE [22]) for building corporate web applications do not follow the scalability best practices and encourage developers to build stateful applications. Therefore, legacy, stateful, coupled or transaction-oriented applications are not addressed in this thesis, as scalability should be taken into account at design time already.

Finally, the economic-based load-balancing mechanism presented in Chapters 4 and 5 is a generic approach that could be applied not only to data partitions or application components, but also to other kinds of resource allocation problems, such as operators placement in a distributed stream processing infrastructure for example.

1.5 Structure of the Thesis

The remainder of the thesis is organized as follows: in Chapter 2 we discuss the problems related to application scalability, formulate best practises to design large-scale, robust, cloud-ready and scalable applications, and give a broad overview of cloud computing and cloud storage. In Chapter 3 we describe and classify the databases available today and discuss the challenges of scaling distributed databases. In Chapter 4 we introduce *Skute*, a self-managed and highly-available key-value store designed to leverage the geographical diversity of resources. In Chapter 5, we propose an economic, lightweight approach for dynamic accommodation of load spikes and failures for composite web services deployed in clouds. In Chapter 6 we introduce *Scalia*, a cloud storage brokerage solution, that continuously adapts the placement of data based on its access pattern and subject to optimizations objectives, such as storage costs. We provide the conclusions of our work and outline future directions in Chapter 7.

1.6 Selected Publications

This thesis is based on the following main publications:

- N. Bonvin, T. G. Papaioannou, and K. Aberer. “Autonomic SLA-driven Provisioning for Cloud Applications”. In 11th *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.
- N. Bonvin, T. G. Papaioannou, and K. Aberer. “An economic approach for scalable and highly-available distributed applications”. In Proc. of the 3rd *IEEE International Conference on Cloud Computing (CLOUD)*, 2010.
- N. Bonvin, T. G. Papaioannou, and K. Aberer. “A self-organized, fault-tolerant and scalable replication scheme for cloud storage”. In Proc. of the *ACM Symposium on Cloud Computing 2010 (SOCC)*, 2010.
- N. Bonvin, T. G. Papaioannou, and K. Aberer. “Cost-efficient and Differentiated Data Availability Guarantees in Data Clouds”. In the 26th *IEEE International Conference on Data Engineering (ICDE)*, 2010.

- N. Bonvin, T. G. Papaioannou, and K. Aberer. “Dynamic Cost-Efficient Replication in Data Clouds”. In First *ACM Workshop on Automated Control for Datacenters and Clouds (ACDC)*, 2009.

The following work is also part of this thesis, but it is still under reviewing:

- N. Bonvin, T. G. Papaioannou, and K. Aberer. “Scalia: multi-cloud adaptive storage”.

Although these publications are not directly related to this thesis, several architectural concepts and distributed applications best practises have been developed and used in this thesis:

- Z. Miklos, N. Bonvin, P. Bouquet, M. Catasta, D. Cordioli, P. Fankhauser, J. Gaugaz, E. Ioannou, H. Koshutanski, A. Mana, C. Niederee, T. Palpanas, and H. Stoermer. “From Web Data to Entities and Back”. In 22nd *International Conference on Advanced Information Systems Engineering (CAISE)*, 2010.
- E. Ioannou, S. Sathe, N. Bonvin, A. Jain, S. Bondalapati, G. Skobeltsyn, C. Niederée, and Z. Miklos. “Entity Search with NECESSITY”. In Proceedings of the 12th *International Workshop on the Web and Databases*, 2009.

Part II

State of the Art

Chapter 2

Distributed Application Scalability

2.1 Introduction

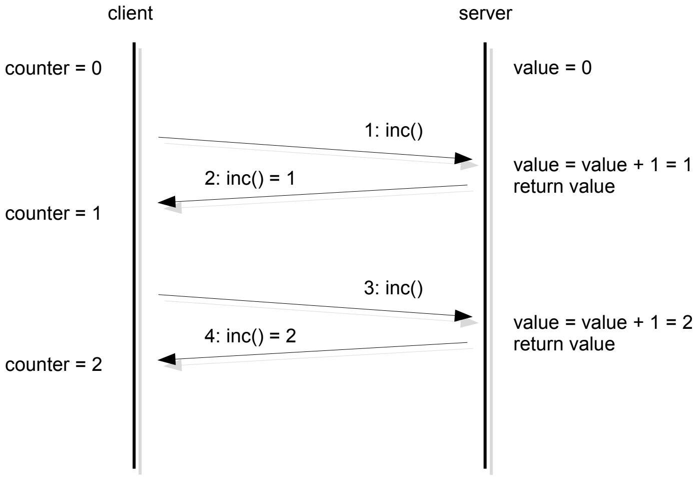


Figure 2.1: Stateful implementation of a function incrementing a value

In Section 1.2.1, a motivating example was introduced where the application layer was composed of web servers. The scalability bottleneck was the database layer and not the application layer: it was assumed that the web servers are stateless, and therefore the application layer can easily scale out by simply adding new servers. However, in practice not every application deployed on an application server is stateless, as this depends on how the application was conceived. Let us consider a very simple application with a single function that increments a value by one, starting from zero. In a traditional implementation, the application stores a local variable initialized at zero and increments it every time the function is called. Figure 2.1 depicts the interactions between a client that increments its local counter and the application providing the increment function. First, the client sends an *incr()* message to the application, which in turn reads its local variable *value*, increments it by one and sends the reply – 1 – back to the client. When the client sends a second request, the application will again increment its local variable and return the result, and so on. This kind of implementation is called stateful as the application keeps a local variable (i.e., a state) holding the current value of the counter.

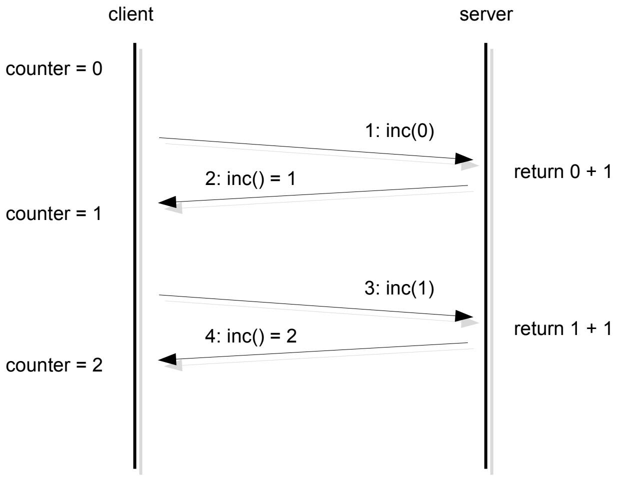


Figure 2.2: Stateless implementation of a function incrementing a value

Another way of implementing the application without requiring to store a state at the server side is to carry the state with the function call, as depicted in Figure 2.2. The client sends the value of its local counter when it calls *incr()*. The application only needs now to increment the received value and sends the result back. There is no need to store anything at the server side. This kind of implementation is called stateless.

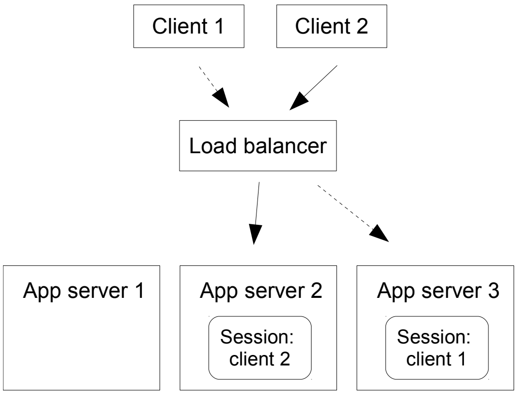


Figure 2.3: Storing sessions locally at the application server

A stateful application keeps a separate state per client, called a session. In order to scale the application layer, a load-balancer is placed in front of several application servers. A common scenario is that each application server stores the sessions of its clients locally, as depicted in Figure 2.3. Obviously, a client has to be served always by the same application server, the one which has stored the session locally. The load-balancer has to be intelligent [1] enough to route the requests of a client always to the same application server. While this setup has the advantage of being easy and cheap to deploy, it does not provide fault-tolerance: if an application server crashes, then all the sessions stored on it disappear resulting in unwanted behaviours and bad user experience. Moreover, hot-spots are unavoidable as the load on a server depends on the volume of the requests: for example, few heavier users may have a large impact in term of resource usage or load-balancers using hash-based routing are unable to properly handle tricky cases such as a large number of users behind a NAT [2] device having the same IP address.

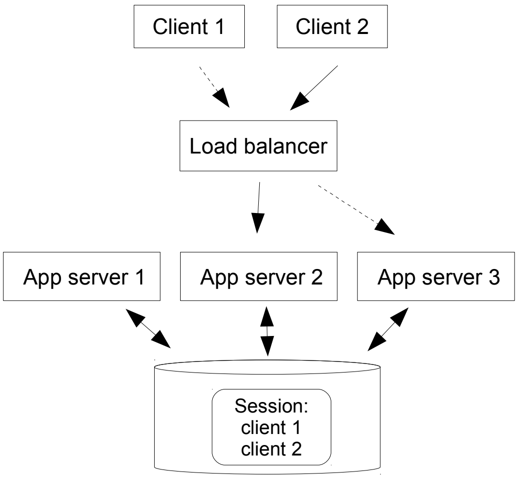


Figure 2.4: Storing sessions centrally

A better architecture that provides fairer load-balancing as well as fault-tolerance of the application layer is shown in Figure 2.4. The sessions are stored centrally in a database or in an in-memory key-value store such as Memcached [26]. A client request can now be handled by any application server. If an application server crashes, the next request can be served by another server as the client session is still available in the centralized store. This approach achieves better load-balancing without creating hot-spots and does not require a clever load-balancer that routes the clients' requests always to the same server. However, the centralized store becomes now the bottleneck and needs therefore to be scalable, as the scalability issue was pushed down the stack to the database layer. The scalability of the database layer is discussed in Chapter 3.

Even if some scalable NoSQL databases (see Chapter 3.3) were developed

[1] This feature is called "sticky session".

[2] http://en.wikipedia.org/wiki/Network_address_translation

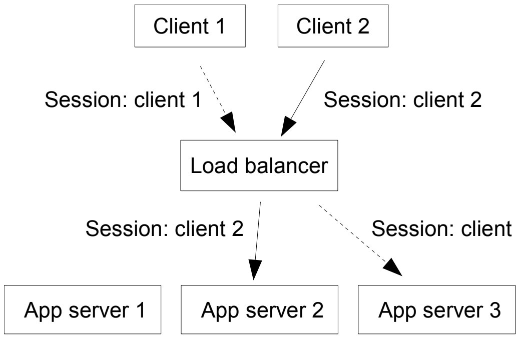


Figure 2.5: Storing sessions at the client side using cookies

specifically to tackle the session case, avoiding completely the sessions remains the best solution, as depicted in Figure 2.5. However, this is rarely feasible in practise due to security constraints and the limited size of the data that can be stored in a cookie or in the URL query string [3].

A monolithic or stateful application will obviously not be able to take advantage of the elasticity offered by the cloud computing paradigm. Section 2.2 will give recommendations to build robust and scalable application, so as to be able to fully exploit the power of cloud computing, while addressing the issues inherent to it.

2.2 Scalability Best Practises

There is no unique recipe or commercial product that will make an application scalable. Every application has different concerns and has to cope with different constraints. However, looking at the infrastructure of successful large-scale applications allows to infer some best practises. In particular, [58, 98, 43, 55] give simple practical recommendations to design and deploy large-scale services. Specific recommendations regarding the elaboration of cloud applications are discussed in [154]. We provide an overview of several architectural best practises to build robust and scalable applications:

- **keep it simple**: keep things simple and robust. This general advice is even more relevant when designing large-scale distributed systems. Complex interactions between the components or complicated algorithms increase the hassle of deploying and debugging the application. Simple things are easier to get right.
- **expect failures**: designing for failures is a core concept of large-scale applications composed of many components. Failures will happen, and often! The application should continue to work without human interaction even if the underlying physical hardware or components fail. For example, Netflix [4], a company relying almost exclusively on a cloud

[3] http://en.wikipedia.org/wiki/Query_string

[4] http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html

computing infrastructure, kills random servers and services in its infrastructure continuously, just to ensure that the application will survive real failures.

- **implement redundancy and fault recovery**: the application should be designed such that any system or component can crash at any time while still satisfying the performance objectives (e.g., service level agreements). A careful architectural design and redundancy are required to achieve four or five nines of availability.
- **implement loose coupling and be stateless**: the application should be split into independent stateless components, so as to minimize the impact of change across dependencies between the components. The functional decomposition groups similar pieces of functionality, while trying to decouple unrelated functionality as much as possible, in order to maximize the flexibility of scaling the components independently of one another. Decoupling also allows components to continue to work, maybe in a degraded mode, when other components have failed.
- **split horizontally**: loose coupling is not enough by itself as the load of a single functional area can surpass the capacity of a single machine. The functional area should be split into manageable, adjustable and fine-grained partitions: for example, by using a look-up table the manageable partitions can be freely moved between the available resources.
- **be asynchronous**: components should be non-blocking and should interact in an asynchronous manner (e.g., through a queue), so that each component can be scaled independently. Moreover, asynchronous flows can better handle load peaks by spreading the load over time with the help of queues, thus decreasing the infrastructure costs compared to a synchronous architecture, which has to be provisioned for the peak load (instead of the average load). For example, approaches like SEDA – Staged Event-Driven Architecture [5] – can be considered to decompose an application into a set of stages connected by queues.
- **avoid distributed transactions**: distributed transactions across various database partitions are slow, costly and not scalable: guaranteeing immediate consistency between the partitions is achieved using a blocking two-phase commit protocol. However, in most large-scale applications, relaxed transactional guarantees are sufficient as discussed in Chapter 3.
- **cache appropriately**: caching slow-changing or read-only data can drastically reduce the load on the infrastructure, and thus the costs. Caching should be used at all levels.
- **audit, profile, monitor and trigger alerts**: every interaction between components should be measured, monitored and should report anomalies. The normal behaviour of the application has to be understood. Some persistent bugs are not reproducible in development or staging environments, so sufficient production data has to be collected and maintained.

[5] http://en.wikipedia.org/wiki/Staged_event-driven_architecture

- **automate everything**: human beings make mistakes all the time and are usually less efficient under stress. So, automating every process (provisioning, deployment, failover, ...) is much more reliable, as automated processes are testable. Moreover, in order to be able to scale, limiting the necessity for human management is essential to keep the human operational expense manageable.

2.3 Cloud Computing

2.3.1 Definition

Being still an evolving paradigm, the definition of cloud computing may change over time. According to the NIST [6], its definition [122] is the following:

> Cloud computing is a model for enabling 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. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.
>
> **Essential Characteristics**
>
> - **on-demand self-service**: a consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider;
>
> - **broad network access**: capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs);
>
> - **resource pooling**: the provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines;

[6] National Institute of Standards and Technology. `http://www.nist.gov/`

- **rapid elasticity**: capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time;

- **measured service**: cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service;

Service Models

- **cloud Software as a Service (SaaS)**: the capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does not manage or 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;

- **cloud Platform as a Service (PaaS)**: the capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations;

- **cloud Infrastructure as a Service (IaaS)**: the capability provided to the consumer is to provision processing, storage, networks, and other fundamental 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 or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

Deployment Models

- **private cloud**: the cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise;
- **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;
- **public cloud**: the cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services;
- **hybrid cloud**: the cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).

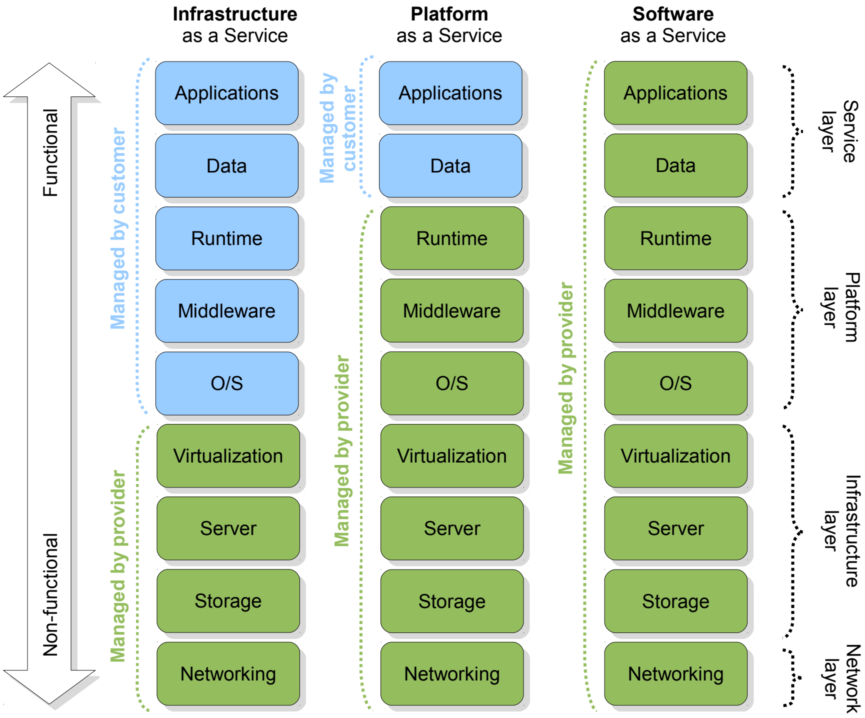


Figure 2.6: Management responsibilities of the different cloud models

Based on the previous definition, it is possible to define the responsibilities of the user and the provider for each service model, as depicted in Figure 2.6.

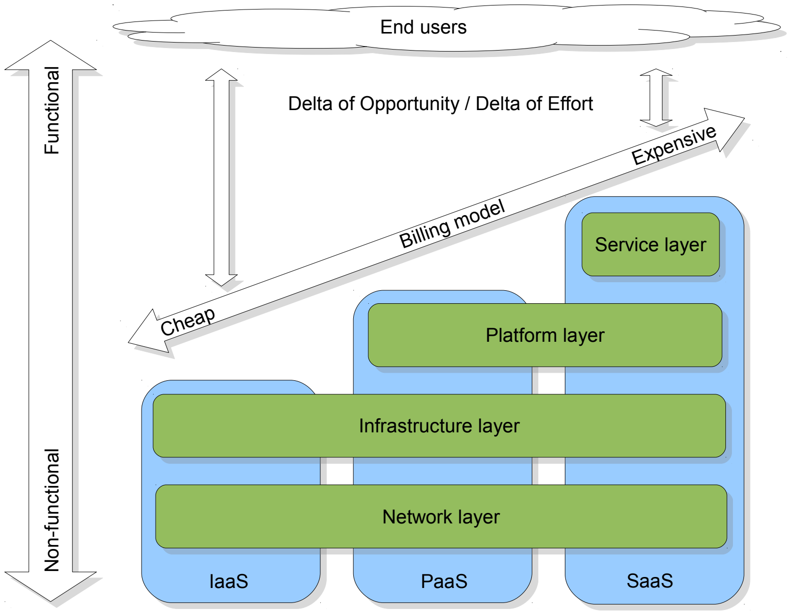


Figure 2.7: Delta of effort and opportunity of the different cloud models

The stack of an application contains four different layers: the network layer, the infrastructure layer, the platform layer and the service layer. An IaaS provider manages the network and the infrastructure layers, that is the networking, the storage, the physical servers and the virtualization. The management of the platform and service layers is left to the customer. In a PaaS model, the provider also manages the platform layer which consists of the operating system, the runtime and the middleware. The customer only needs to manage its applications and the corresponding data. Finally, in a SaaS model, all layers are managed by the provider, the customer only consumes the services.

There exists a delta of effort and a delta of opportunity between the three models IaaS, PaaS and SaaS as shown in Figure 2.7. The delta of effort quantifies the amount of skills, abilities, experience and technology that a user has to provide in order to deliver a fully functional service. The delta of opportunity represents the area in which the user can innovate and differentiate against its competitors. Both the delta of effort and the delta of opportunity decrease when going from the IaaS model to the SaaS model.

2.3.2 Benefits

With the traditional deployment model where most physical resources such as servers, network devices and storage systems are directly managed by the user, the latter stays in full control of the whole system, allowing to focus on data consistency and to ensure easier sharing and reuse of data and components, stronger privacy and security as well as customizability. However, this model does not take advantage of the economies of scale and the elasticity offered by the cloud computing paradigm, which provides easier provisioning of resources, focuses on data availability (rather than consistency) and allows to more directly reach regional as well as global markets, as cloud providers are present all around the globe. Cloud computing providers manage a large amount of low-cost computers using automated maintenance and system management tools. Thanks to the automation, only a few administrators are sufficient to manage thousands of computers in a simple and cost-effective manner. Moreover, virtualization techniques make multiple operating systems running on the same computer possible, allowing a flexible and efficient use of the computing resources.

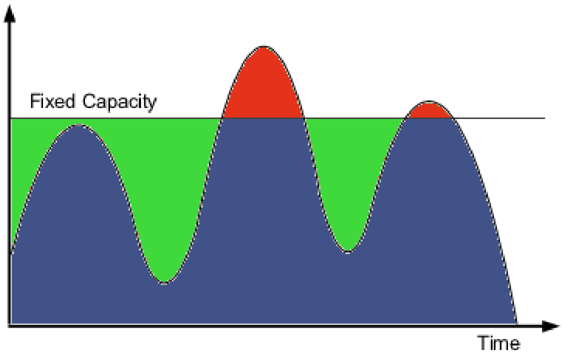


Figure 2.8: The problem with fixed capacity: the capacity is either wasted (in green) or insufficient (in red) according to the demand (in blue)

Predicting how much capacity will be needed as demand changes and planning the required resources to cope with application load remains an important and difficult problem. For example, a startup company providing online services that suddenly get very popular, needs quickly more computing power or storage to serve the increasing load. The company should react rapidly not to lose customers. Similarly, to handle flash-crowd, the underlying infrastructure has to be flexible and reactive. With cloud computing it is now possible to automatically use as much computing or storage resources as needed, on demand without the need of over-provisioning or the risk of under-provisioning. Let us consider an application with a traffic demand varying over time, as depicted by Figure 2.8. With a fixed capacity, resources are either wasted when the demand is below the allocated capacity (in green in Figure 2.8) or insufficient during load peaks (in red in Figure 2.8). Thanks to the new cloud computing solutions, a user pays only for the resources consumed while being always able to satisfy the demand.

Cloud computing presents several benefits, explaining why a growing amount of companies embrace it:

- **efficiency**: cloud computing has the potential to lower hardware and IT costs by removing the need for a company to buy hardware and to maintain its own small data center by a dedicated team, having to take care of power, cooling, redundancy, etc. Concerning private clouds, virtualized applications can run on top of a pool of resources, requiring less hardware and thus less staff.

- **agility**: resources can be added and removed at will very quickly, allowing to survive load peaks or hardware failures more easily.
- **flexibility**: pay only for what you consume. Hosted services are billed on a monthly basis, while infrastructure and platform services are usually billed based on the capacity used.

A well designed application that can take into account new resources automatically will largely benefit from the cloud computing paradigm, allowing to support a fast growing amount of traffic without pain. However, this novel paradigm is not perfect yet and requires several improvements, especially with public clouds: data security, confidentiality, privacy, support or vendors' lock-in are still a major concern.

2.3.3 Cloud Storage

Similarly to provisioning processing resources on demand (i.e., virtual machines), providing storage resources on demand is also a fundamental part of the IaaS service model. Cloud storage enables a new way of storing and distributing data, where the data owner is billed based on its usage with a minimal commitment. Among the advantages of cloud storage we can cite the predictable storage and delivery costs, the ability to access data from anywhere with many different kind of devices using standard protocols (e.g., HTTP) and the almost infinite capacity offered by these new solutions. Famous cloud storage providers include Amazon S3 [2], Google Storage [16], Microsoft Azure [27], RackSpace Cloud Files [42] or EMC Atmos [12]. Similarly to cloud computing, a big advantage of cloud storage is the ability to deliver on-demand capacity, avoiding the need for capacity planning.

Storage Interface

There are three fundamental ways to access and interact with a network storage:

1. **block storage**: block storage systems are accessed using standard protocols such as iSCSI [20] or AoE [106]. A storage area network (SAN [7]) typically provides block-based storage, where file system concerns are left on the client side. It provides high data throughput using direct access to the data at disk or fibre channel level. A SAN appears as a disk which has to be formatted and mounted by the client operating system.
2. **file storage**: it is accessed by network file sharing protocols such as CIFS [9] or NFS [140]. Network-attached storage (NAS [8]) provides storage as well as a file system. A NAS appears as a file server to the client

[7] `http://en.wikipedia.org/wiki/Storage_Area_Network`

[8] `http://en.wikipedia.org/wiki/Network-attached_storage`

operating system and multiple clients can access the data at the same time.

3. **object storage**: object storage encapsulate the data, including attributes and metadata, allowing to determine data layout on a per-object basis, thus improving flexibility and manageability. It is typically accessed through a REST [86] or SOAP [47] API. Unlike block or file storage, clients usually access the data remotely over Internet. This is the interface offered by the cloud storage providers.

Cloud Storage Gateway

Storing and retrieving data from a cloud storage provider requires programming skills in order to interact with the provider's proprietary API. Moreover, every provider has its own API which makes difficult and costly to use several providers at the same time. As cloud storage providers provide only an object storage interface, a file or block interface is required to allow existing applications to use this new kind of elastic storage: Cloud Storage Gateways allow to access several cloud storage providers through a traditional file or block access, removing the need to use the providers' proprietary API, as depicted in Figure 2.9. A local Cloud Storage Gateway comes usually as an appliance which not only provides a transparent access to multiple cloud providers, but also offers advanced features such as caching, encryption, compression or deduplication [121] for example.

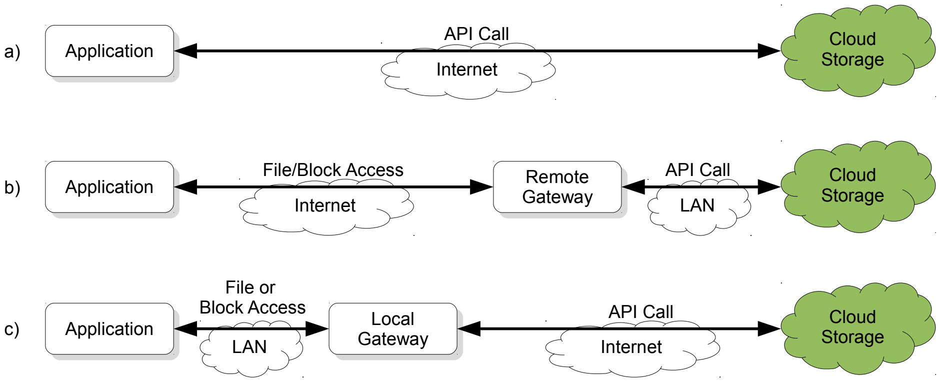


Figure 2.9: Access to Cloud Storage: *a)* direct access *b)* access through a remote gateway *c)* access through a local gateway

Content Delivery

Cloud storage is mainly used to store large amount of static files like pictures or read-mostly data. To deliver data quicker to the end user, the data should be stored geographically close to the consumers. If the end users of a photo-sharing website are mainly located in Europe, it makes sense to choose a cloud storage provider also located in Europe. However, most of the time

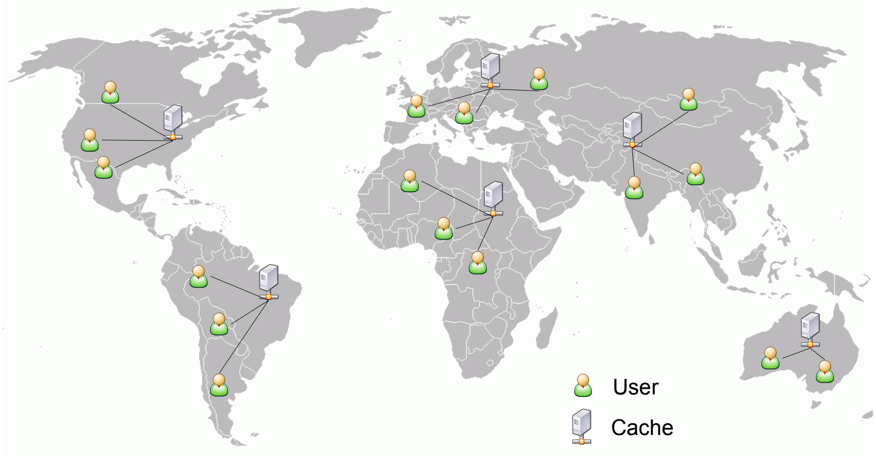


Figure 2.10: Content Delivery Network

the visitors of a web site do not come from a single region. In order to serve end users' requests efficiently, the data should be cached close to its destination. Large cloud storage providers also offer a worldwide caching system working seamlessly with their storage solution: a content delivery network (CDN) [104].

A CDN is composed of multiple computers interconnected on Internet providing content efficiently to end users by replicating the content on those computers and by redirecting the users to the geographically closest copy of the data, so as to minimize the end-to-end latency, as depicted on Figure 2.10. That is, the same data exists in multiple locations cleverly dispersed over the world, allowing to serve the data to a large number of end users reliably even during sudden demand peaks.

Beside reducing the load of the server hosting the original data and minimizing the latency for end users, a CDN technology also provides some sort of content redundancy: if the origin server has a failure, the content is still cached and accessible to the users, allowing the application to work in a degraded mode.

2.4 Conclusion

Cloud computing offers a lot of new opportunities for small companies as well as for big players, allowing to build an application that can grow from hundreds to millions of clients in a very short time span. In Chapter 5, we describe a framework to build highly-available and scalable applications, which are able to take advantage of the elasticity of the cloud computing infrastructure while coping with unreliable resources. Cloud storage and CDN technology permit to store and deliver a huge amount of data efficiently, reliably and in a cost-effective manner. A non-negligible part of the global Internet traffic is already served by CDNs: for example, Akamai [9], a leading CDN provider, is known to deliver between 15% and 20% [126] of all Internet traffic worldwide. With the rise of cloud storage and their coupled delivery offers, the percentage of data served by content delivery networks will probably keep on growing.

However, several issues need to be tackled before a widespread adoption of cloud computing can happen, especially regarding data governance, security and business environment. The ownership of a data stored in a cloud infrastructure managed by a third-party or the right to access it are questions related to data governance. In the case of legal dispute between the provider and the data owner, which is the legal jurisdiction? Data privacy and confidentiality are also a primary concern for many cloud users. Security is another big challenge: data stored in a cloud should be always available, but only to the authorized people. A large amount of work has still to be done in order to ensure only authorized access, integrity and availability of data. Definitive deletion of data should also be ensured by the cloud providers. Finally, interoperability between cloud providers and data portability is mandatory to drive competition so as to increase the resilience and the maturity of the cloud computing ecosystem. The market is still reticent about cloud storage, mainly because transferring data between cloud storage providers is still a major problem, as there is no peering agreements between providers. Today, there is no standard allowing to move data around, thus the customers are locked in by vendors: switching to a new provider involves a prohibitive cost. In Chapter 6, we describe an approach able to avoid the vendors' lock-in, while storing content to the optimal set of cloud storage providers so as to minimize the costs.

[9] `http://www.akamai.com/`

Chapter 3

Database Scalability

3.1 Introduction

In Chapter 2, we have discussed the scalability challenges occurring at the application layer, along with the related opportunities offered by cloud computing in general and cloud storage in particular. In this Chapter, we will discussed the challenges to reliably store and query information in a scalable manner.

As non trivial applications require a database to store important in a persistent manner, the database layer should also be scalable in order to handle the load generated by the application layer. In the current corporate computing architectures, the database layer is mostly composed of relational database management systems (RDBMS) [1], which provide a set of properties guaranteeing that transactions are processed reliably. The main characteristics of a reliable transaction system [92, 93] are known under the acronym ACID [97]:

- **atomicity**: this property prevents that a series of updates to the database occur only partially, which is more problematic than entirely rejecting the whole series. That is, a series of database operations, called a transaction, either all occur or nothing occurs.
- **consistency**: this property ensures that any transaction the database performs will take it from one consistent state to another. It ensures the correctness of the database, and states that only valid data will be written to the database.
- **isolation**: this property defines how/when the changes made by one operation become visible to other concurrent operations. It requires that other operations cannot access data that has been modified during a transaction that has not yet completed.
- **durability**: this property guarantees that the results of transactions that have committed will be stored permanently even in case of any kind of system failure.

[1] http://en.wikipedia.org/wiki/Relational_database_management_system

The databases following the relational model are able to support a large range of applications, but are at the same time complex and come with performance constraints. Many applications do not require such a rich programming model, but will benefit from lighter, simpler and more scalable data stores. With the rise of very large web applications such as Amazon.com, Facebook or Google, new classes of data stores have been developed to store and fetch key-values pairs very efficiently, to store and process huge amount of data or to deal with large graphs for example. Structured storage systems can be classified [99] into several categories based on the underlying goal an application wants to achieve:

- **feature-first**: a relational database management system, such as Oracle Database [39], MS SQL Server [28], IBM DB2 [19] or MySQL [31] is the natural choice to support typical corporate applications such as enterprise resource planning (ERP) [2], human resource management systems (HRMS) [3] or customer relationship management (CRM) [4]. Although these softwares can be demanding and business critical, they usually run on a single database instance and thus only vertical scaling is considered, even in large companies.

- **scale-first**: for very large applications such as Amazon.com, Facebook or Google scaling the database layer is more important than the features. As such applications do not run on a single database instance, scalability is accomplished either by horizontally partitioning the dataset (see Section 3.2.2) or by using a scalable NoSQL datastore (see Section 3.3), such as Apache Cassandra [4], Project Voldemort [40] or Apache HBase [18].

- **simple structure storage**: a large amount of applications do no require the features of a relational database management system, let alone its costs and complexity. They also do not have the scalability requirements of the scale-first class of applications; they just need a structured storage that is simple, cheap, easy to manage, fast and that can process simple queries efficiently. Data stores such as Berkeley DB [5], which is an embedded key-value database, CouchDB [10] or MongoDB [30] typically fall into this category.

- **purpose-optimized stores**: existing relational database management systems are not optimized for many classes of applications, such as data warehousing, stream processing, RDF processing, analytics for scientific research or time series processing. Many databases have been developed by companies focusing exclusively on specific market segments such as OpenTSDB [38], VoltDB [51], SciDB [46] or Hadapt [17].

It distinctly appears that no unique solution is appropriate for all kinds of problems or applications: it is important to find the suitable solution for a given problem, and therefore both relational and non-relational structured storage systems are essential and should coexist.

[2] http://en.wikipedia.org/wiki/Enterprise_resource_planning

[3] http://en.wikipedia.org/wiki/Human_resource_management_system

[4] http://en.wikipedia.org/wiki/Customer_relationship_management

3.2 Relational Databases

The first approach to scale a relational database is to take advantage of the replication mechanisms, as will be described in Section 3.2.1. However, as shown in Section 1.2.1, replication cannot scale beyond a certain point. To further scale the database layer, it is required to partition the database (as will be described in Section 3.2.2).

3.2.1 Replication

The various modes of replication expand the read capacity over multiple servers, by keeping multiple replicas of the data on multiple servers. However, the write capacity cannot be scaled satisfyingly with this method.

Master-Slave Replication

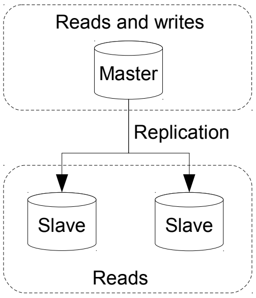


Figure 3.1: Master-slave replication

In a master-slave replication setup, all write operations, such as insert, update, delete, create or alter, are performed on the master server. All changes are then replicated asynchronously to the slave servers. In the example depicted in Figure 3.1, the read capacity has grown by three, as all servers can handle read requests. In order to further scale the read capacity, new slaves simply need to be added to the setup.

However, beside being a single point of failure, the master server remains a bottleneck, as all write operations have to be handled by it, before being replicated to the slaves. A major problem with this setup was pointed out in the motivating example of Section 1.2.1: as every server has to perform every write operation, the master-slave setup cannot scale the write capacity and eventually, if the write load increases, the read capacity will be absorbed by the write load. Finally, a problem specific to asynchronous replication occurs when network traffic increases: when replication is delayed, slave servers may return stale data. Replication lag, which corresponds to the time it takes for an operation executed on the master to be executed on a slave, is usually low (in the order of milliseconds) for an idle system, but can be problematic when the server load increases as the application functionality may be impaired.

Tree Replication

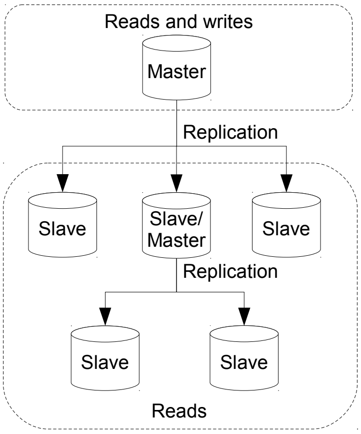


Figure 3.2: Tree replication

In a master-slave setup with a large amount of slaves connected to a single master, the resources requirement (such as bandwidth for example) of the master becomes considerable. Consequently, to limit the number of slaves connected to a single master, a replication tree [102] can be created by turning some slaves into a master of further slaves, as shown in Figure 3.2. Additionally, the tree replication allows to replicate only a portion of data from a master to its slaves, reducing data traffic and creating specialized slaves. When a specialized slave replicates only a single table, its capacity can be increased by dropping unused indexes or by changing the underlying storage engine (e.g., using MyISAM instead of InnoDB in the case of MySQL.)

However, this replication scheme involves greater replication lag as shown in Figure 3.2: a write operation is first executed on the master and then replicated to the second-level slaves. Once the operation is performed on the second level, it gets replicated to the third level, and so on until the update reaches all bottom slaves. Another drawback of this approach is that the bottom slaves will go stale if a middle-tier slave encounters a failure, which can lead to inconsistencies impacting the application functionality.

Multi-Master replication

In a multi-master replication [102, 111] scenario, the data is stored by a group of computers (i.e., the master database servers) and updated by any member of that group. In case of a master failure, other members of the group are still able to update the database. Transactions can be propagated among masters using two ways: synchronously or asynchronously.

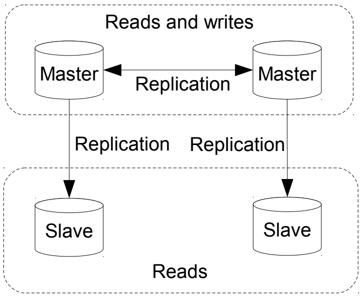


Figure 3.3: Multi-master replication

Synchronous replication increases communication latency and involves advanced techniques such as 2 phase commit [60, 91, 131, 120], 3 phase commit [146], Paxos [114] or various approaches to state machine replication [70, 112]. According to the protocol or the number of faulty master tolerated, performance and availability of a multi-master setup can be less than the ones of a single master setup. In addition, the scalability of this approach is limited because every write operation has to be acknowledged by a minimum number of master database servers in order to reach a quorum [153, 88] such that consistency of the data can be guaranteed. For example, in the case of state machine replication with non-byzantine failures [116], the group of master database servers should be composed at least of $2f + 1$ members [115], where f is the number of faulty servers.

Asynchronous replication is more popular due to its advantages over synchronous replication. First, it uses less network bandwidth and provides higher performance, by grouping the transactions rather than propagating each transaction separately. Second, it is more resilient to failures: if a master server crashes, other master servers still can execute the updates. However despite all its advantages, asynchronous replication has several drawbacks: concurrent updates on two different master servers can lead to conflicts and data inconsistencies. As a simple workaround to avoid inconsistencies, the application interacting with the database should never write the same entry on two different servers at the same time, by using for example a third party locking mechanism such as ZooKeeper [107]. Moreover, in case of a large number of concurrent write operations to the database, no single "valid" copy of the data exists: entries written on one master server will take some time to get replicated to the other master servers, that is, each master database server may have a different copy of the data at a given point of time.

3.2.2 Database Sharding

As discussed in the previous section, replicating data only permits to scale the read capacity. In order to also scale the write capacity, data needs to be split into partitions, or *shards*, which may be spread over multiple independent servers with their own resources, such that partitions have dedicated

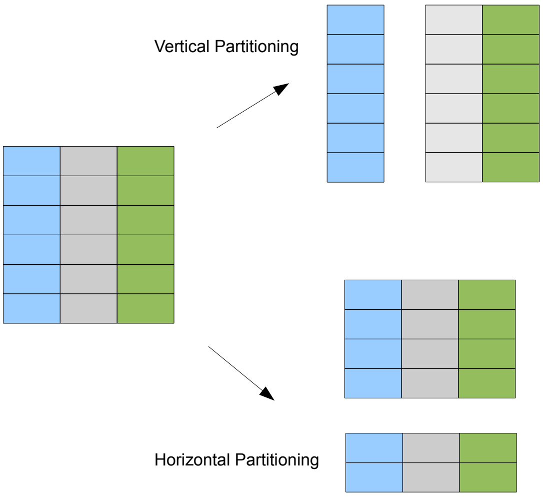


Figure 3.4: Database partitioning: horizontal versus vertical

write capacity. That is, the total write capacity of a database is directly related to the number of its partitions.

There are two basic ways of partitioning the data as shown in Figure 3.4. The first approach, known as vertical partitioning, is to split the attributes by creating tables containing fewer attributes (i.e., fewer columns) and making use of supplementary tables for the remaining attributes. While vertically partitioning the data is relatively easy to set up, it only offers limited scalability. The second approach, called horizontal partitioning, splits the tuples (i.e., the rows) into different tables, allowing to scale to any number of partitions.

While partitioning is also used to improve the performance of a database server within a single instance of a schema, by reducing index sizes for example, we discuss it here in the context of scalability, where multiple instances of the schema exist, allowing the load to be split across multiples servers. Database sharding is directly related to shared nothing architectures [148] and goes beyond horizontal partitioning: as partitions are spread over multiple independent servers, potentially hosted in different datacenters, the load of a partitioned table is also spread over multiple servers, not just over multiple indexes like in the case of horizontal partitioning.

Vertical Partitioning

The main idea is to put on different partitions tables belonging to different functional areas, such as business units (like administration, sales, marketing, production, finance, . . .), or belonging to different functional aspects of an application: for example in a SOA-based application, services are in charge of independent parts of the application, each providing a separate functionality, and will therefore probably access distinct tables of the database. Both the data and load scalability are driven by the functional aspects of the application. The latter has to be modified to pick the right partition depending on the tables that are queried. Moreover, the join operations spanning several partitions have now to be moved to the application level, increasing the complexity and the load of the application. The database is no more truly relational, and the scalability is limited because a functional aspect of an application cannot grow over the capacity of the underlying database server hosting the partition.

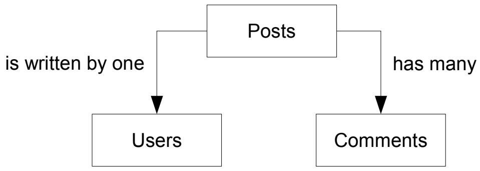


Figure 3.5: Simplified data model of a weblog application

Figure 3.5 presents a simplified database schema of a web application allowing an author to post blog entries, which anonymous readers can comment. The database contains only three tables: *users* used to store the details and credentials of the authors of a blog, *posts* which contains the blog entries posted by the authors and *comments* which stores the comments attached to a blog entry. Let's imagine that the current database is no more able to process the queries and thus its capacity needs to be increased. After having analysed the application, it appears that the table *posts* has grown too much and requires too many resources. A solution is to place *posts* on a second server, while *users* and *comments* stay on the initial server, as depicted on Figure 3.6. Although the scalability issue has been resolved by vertically partitioning the database, moving *posts* to another cluster broke the relational model: the comments attached to a blog entry are no more linked by a relational constraint, as they are now on a different partition. So, if a blog entry is deleted, the application should now take care of deleting the related comments. Previously, this action was handled automatically by the database (e.g., using cascading triggers).

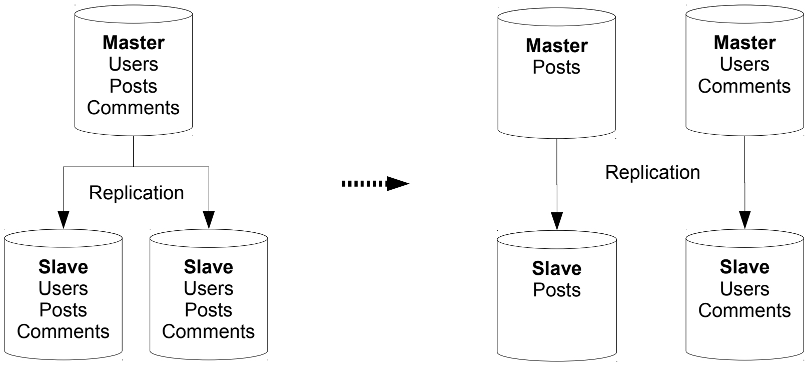


Figure 3.6: Vertical partitioning: as the table *posts* needs additional capacity, it is moved to a different partition

What happens if the table *posts* keeps growing beyond the capacity of the server hosting the partition? This scalability barrier can be solved by splitting the table into two or more tables; this technique, called horizontal partitioning, is described in the next subsection.

Horizontal Partitioning

Horizontally partitioning the data is the most general approach to scale out a database to arbitrary sizes. Instead of splitting tables based on functional aspects, the tuples of a table are now split by key into several smaller tables (in term of rows), each of them being hosted on a different partition. When the size of the dataset or the amount of queries to the database increases, the number of partitions is also increased so as to keep a similar amount of data and queries per partition: adding new servers permits to increase the read and the write capacity of the database, as the global load is spread over a greater number of servers.

This partitioning scheme comes with a non negligible cost in term of complexity: a range query on a table spread over multiple partitions requires to fetch data on all involved partitions, merge and sort the data to obtain the final answer to the query. Things are getting even more complicated with a join operation involving multiple tables located on several partitions. Queries across partitions can be circumvented using denormalization [145, 141, 160], that is, by grouping or by adding redundant data, so as to be able to fetch related data from the same partition. For example, tables with data that is not frequently updated such as a list of countries could be easily duplicated on each partition to avoid queries across partitions. Finally, a layer responsible for handling the partition access logic needs to be created, further increasing the complexity of this approach.

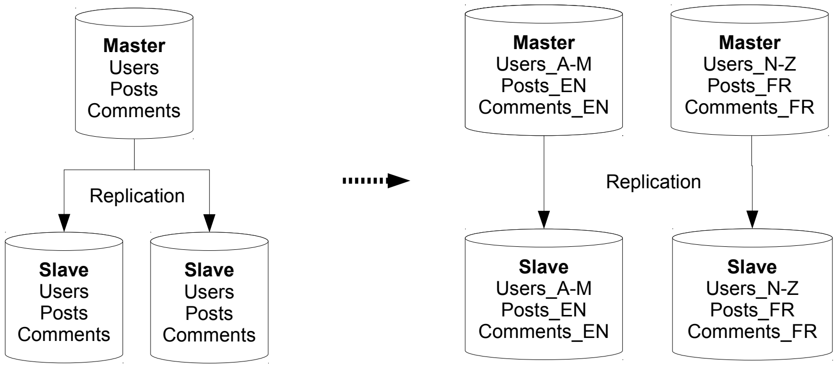


Figure 3.7: Horizontal partitioning: tables are partitioned into smaller tables according to a key, such as the language of the blog posts or the name of the authors

Figure 3.7 shows a possible partitioning of the database: the table *posts* is now split into two tables *posts_FR* and *posts_EN* containing the posts written in French and those written in English (we consider in this example that half of the writers are French speaking, and the other half are English speaking).

In order to avoid cross-partitions queries, the comments related to a blog entry are hosted on the same partition. Finally, to balance the load between both partitions, the table *users* is split based on the first letter of their names.

The decision how to split a dataset depends on the nature of the application and how the data will be queried or accessed. There exists three main ways of partitioning the data:

- **hash based**: the dataset is split according to a hash function. A simple approach is to use the modulo function with the number of partitions, as depicted in Figure 3.8: when a new partition is added to further scale out, the data will have to be rebalanced, which is a complicated and costly operation. However, using a consistent hashing mechanism [110] mitigates the amount of data that needs to be moved, by using virtual partitions.

- **range based**: the dataset is split according to several ranges. In the case of the table *users*, the users with their name starting from ’a’ to ’m’ go to the first partition and users with their name starting from ’n’ to ’z’ go to the second partition. Other kind of ranges can be imagined: a time range can be used to partition the table *posts*: entries written between 2000 and 2009 go to a first partition while entries between 2010 and 2019 go to another partition.

- **directory based**: the dataset is split arbitrarily and a level of indirection is added so as to map each partition key to a partition. Altering the mapping, which is stored in a directory, allows to move the data corresponding to a partition key to a different partition easily. Although this partitioning scheme is the most flexible, it introduces overhead, makes more difficult for an application to query the database and may create a single point of failure.

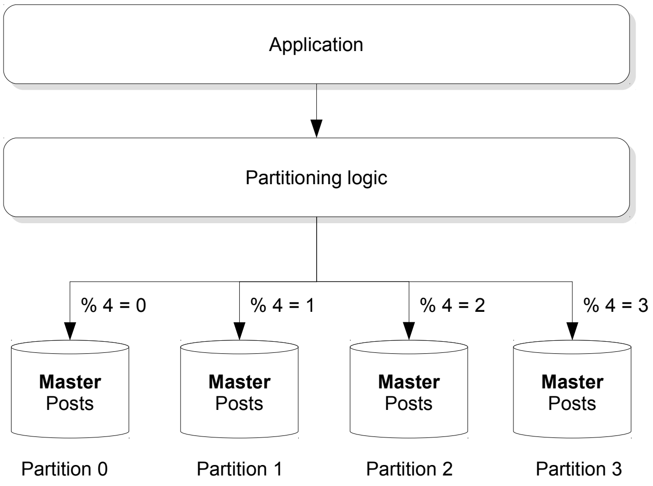


Figure 3.8: Horizontal partitioning: a layer manages the logic to access the partitions

Discussion

Sharding a database results in the dataset being fragmented and spread over multiple separate servers, reducing the advantages of the relational model. This approach might cause more problems than it solves. If the relational constraints on the data are not needed or not available due to database sharding, then a traditional relational database might be unnecessary; in that case a NoSQL or NewSQL database is easier to scale. It also worth to mention that in practice relational databases scale vertically (i.e., by adding resources such as memory and CPU to the database server) substantially more than commonly admitted by many specialists [143]. Moreover, aggressive caching at every layer (client, application, database, . . .) greatly reduces the read load of the database, and should be therefore considered before using advanced techniques such as sharding.

3.3 NoSQL Databases

The term *NoSQL* standing for "Not only SQL" was popularised in early 2009 and is an umbrella term to define a class of non-relational structured storage systems that differ from classic relational database management systems by focusing on specific niches such as scalability or graph processing for example. The rapid development of NoSQL databases was partly influenced by two key research papers:

- **Google BigTable** [71] defines a specific data model focused on storing and querying multi-column data and uses a range-based partitioning scheme which allows data to be stored on multiple distributed nodes;
- **Amazon Dynamo** [78] uses a simpler key-value data model, but the system is more resilient to failures, thanks to a looser consistency model.

Although there is today no widely accepted definition, a NoSQL data store has some of the following properties: non relational, distributed, (horizontal) scalable, schema-free, easy replication support, simple API [5], big data [6] ready, eventually consistent.

While Figure 3.9 gives a gross overview of the data models of the main NoSQL databases available, they might be better classified using a four dimensional representation: query model vs. data model vs. distribution model vs. disk/memory data structure. A survey of existing NoSQL approaches is presented in [151].

NoSQL data stores were primarily developed with a strong focus on scalability by making some engineering trade-offs described by the CAP theorem, which was introduced in [68] and formally proven in [89]. It states that a distributed system can satisfy any two of the following guarantees at the same time, but not all three:

[5] http://en.wikipedia.org/wiki/Api

[6] http://en.wikipedia.org/wiki/Big_data

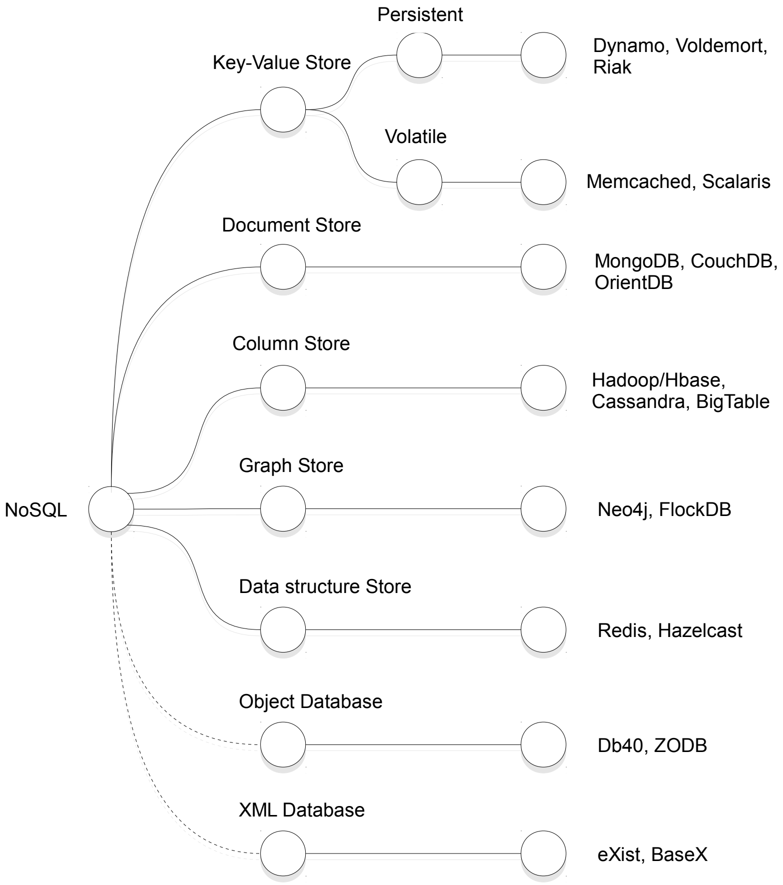


Figure 3.9: NoSQL databases landscape

- **consistency**: all clients see the same data, even in presence of concurrent updates. The definition given in [89] defines consistency as being "equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time".
- **availability**: all clients are able to access some version of the data, even in the presence of failures. In [89], it is defined as "every request received by a non-failing node in the system must result in a response."
- **partition tolerance**: the system continues to operate despite arbitrary message loss: the system properties hold even when the system is partitioned. However, no definition of partition tolerance is given in [89], which has led to some confusion [135].

This theorem is sometimes used inappropriately as the justification for giving up consistency [149]. The problem is that the concept of consistency in ACID does not describe the same concept as consistency in CAP [162]. In the proof [89] of the CAP theorem, consistency means that every object is linearizable [103]: if a client gets a response, it has the right answer, de-

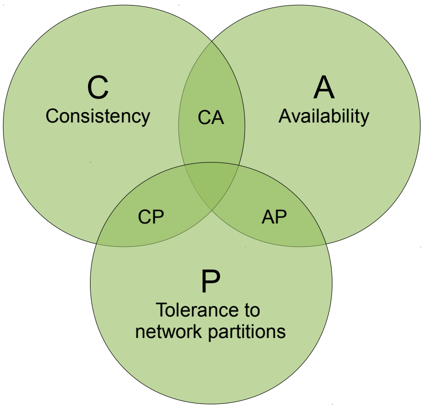


Figure 3.10: Brewer's Conjecture: CAP theorem

spite concurrency. Linearizability, or one-copy serializability [64], ensures that operations are serializable and applied in a consistent order. In fact, it corresponds to the concept of isolation from concurrent operations in ACID and not consistency in ACID. Moreover, in the proof of the CAP theorem, the concept of transaction is not tackled, where a transaction, which may commit or abort, has a start, a sequence of operations and an end. Contrary to CAP, ACID refers to an entire transaction.

As depicted in Figure 3.10, there are three choices offered by the CAP theorem:

- **CA**: consistent and available, but not partition tolerant; HBase and BigTable are notable examples of CA systems.
- **AP**: available and partition tolerant, but not consistent; Amazon Dynamo and Cassandra belong to the AP systems for example.
- **CP**: consistent and partition tolerant, but not available;

The last kind of system is consistent and partition tolerant (CP), but not available: a system that is never available is completely useless. In fact, a system that is consistent and partition tolerant (CP) sacrifices availability only in case of a network partition. An available and tolerant to partitions system (AP) sacrifices consistency all the time, not just in case of a network partition. Availability and consistency in the CAP theorem are therefore asymmetric [54], where some properties apply in general while others apply only when there is a network partition. The asymmetry resides in the probabilities of different failures. The lesson of the CAP theorem is that if there is a possibility of network partitions, then availability and consistency cannot be achieved at the same time: the choice is almost always between consistency and availability.

However, the trade-off between consistency and availability may give a wrong perception that a system has to give up consistency in order to get availability. Even if a system has to give up only one property, some systems such as PNUTS [73] want to give up both consistency and availability at the same time, in order to decrease latency. First, PNUTS gives up availability: if the master replica of a data is unreachable, then the data can no more be updated. Then, regarding network partitions, keeping the consistency between two replicas located at two geographically distant places is costly in term of latency, and might be undesirable in many applications. So, to reduce latency, the replication is done in an asynchronous manner, which in turn reduces consistency. Under normal circumstances, PNUTS gives up consistency for latency, and when a network partition occurs, it gives up availability rather than additional consistency.

PACELC [54] is an evolution of the CAP theorem that takes into account the asymmetry of availability and consistency as well as the impact of latency: if there is a network partition (P), a system may choose between availability (A) or consistency (C) else (E) a system may choose between latency (L) or consistency (C). Using this model, systems such as Cassandra or Dynamo are labelled as PA/EL, while full ACID compliant database systems can be seen as PC/EC systems. Finally, PNUTS is an example of a PC/EL system. Even if PACELC goes in the right direction to properly define distributed systems, the existence of PC/EL is debatable [163] and somehow confusing.

Although most of the NoSQL data stores are scalable and capable of managing a large amount of data, highly skewed popularity of data items is rarely taken into account properly, which can have a severe impact on the global performance of the data store. Moreover, data being highly valuable, a data store must ensure that data will be always available, despite failures. Data should be replicated in several geographical distinct places, so as to survive to major failures (e.g., a datacenter failure). In Chapter 4, we improve a Dynamo-based data store, such that popular data items are given more resources to handle the load, while data replicas are geographically spread to avoid correlated failures.

3.4 NewSQL Databases

The term *NewSQL* [36] defines relational databases designed for scalability while preserving the structured query language (SQL), the relational model and the ACID properties, contrary to NoSQL databases which do not provide all the features of the traditional relational database systems. These databases, such as ScaleBase [44], GenieDB [15], NimbusDB [37], Xeround [53], VoltDB [51] or Schooner [45], try to improve the relational database model in reaction to the success of NoSQL databases. The term NewSQL being misleading, [36] gives the following explanation:

> And to clarify, like NoSQL, NewSQL is not to be taken too literally: the new thing about the NewSQL vendors is the vendor, not the SQL.

They can be divided in several categories:

- **new MySQL storage engines**: besides scaling very well, this category offers the same programming interface for MySQL users. The obvious limitation is that only MySQL is supported. Xeround or Schooner are good examples of databases in this category.

- **new databases**: completely new databases are designed to enable scalability while offering features of the traditional relational databases. Some examples of this category are VoltDB or NimbusDB.
- **transparent sharding**: instead of extending an existing database or creating a new database to support the scalability requirements, transparent sharding addresses the complexity of sharding and provides a runtime infrastructure to make the shards appear as a single database to applications. Being database agnostic, this solution allows to reuse existing database systems. ScaleBase and dbShards [11] are examples of this category.

Similarly to NoSQL databases, each NewSQL solution addresses specific needs.

3.5 Consistency Models

The main approaches to data consistency in NoSQL databases are strong consistency where all replicas remain synchronized, and eventual consistency where replicas are allowed to get unsynchronized, but will eventually be synchronized with each other.

Let's consider a distributed database consisting of N nodes, where a data d is stored. According to [155], three quorum parameters are important:

- R: the number of replicas of d to read from;
- W: the number of replicas of d that acknowledge the write operation before the operation completes;
- S: the number of nodes that store replicas of d.

As long as $R + W > S$, strong consistency is ensured because the read set and the write set always overlap on at least one element. When $R + W \leq S$, the read set and the write sets might not overlap, hence only eventual consistency can be guaranteed.

The values for R and W have a direct impact on the availability or the performance of the database. If $W = S$, then a write operation has to be acknowledged by all S replicas and thus will fail if one of the S replica is unavailable. On the other hand, we can now set $R = 1$ which provides very good read performance.

3.5.1 Strong Consistency

The goal of strong consistency is to ensure that all replicas hosting the value of a given key will always reach a consensus on that value. That is, all clients querying the database have to see the same view, even in presence of concurrent updates: every read has to return data from the latest successful

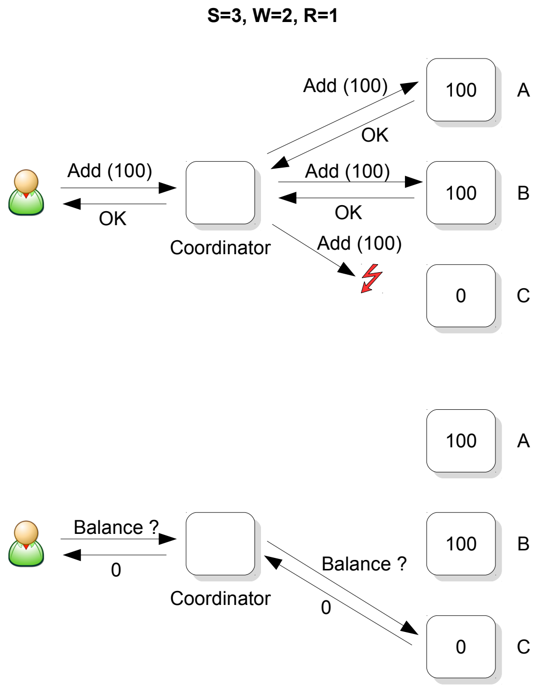


Figure 3.11: Only eventual consistency can be ensured when $R + W \leq S$

write. There are basically two ways to achieve this: either, all operations (read, write, delete) for a given key have to be executed on the same unique node, or a distributed transaction protocol is required such as Paxos [114] or two-phase commit [60, 91, 131, 120]. Note that consistency cannot be achieved at the same time with availability and partition tolerance as seen in Section 3.3.

As an example, consider a simple banking system where an account, represented by its own key, has 3 replicas (i.e., $S = 3$) and an initial balance of 0 $. The account *myaccount* is thus replicated on 3 different nodes, namely A, B and C. Let's deposit 100 $ into *myaccount* and directly check the balance to verify that the money is really on the account. On Figure 3.11, the write operation is successfully performed by nodes A and B, but node C did not get the message, due to a temporary network failure for example. As $W = 2$, the coordinator confirms to the user that the write operation was a success. When the user wants to check the balance of the account, as $R = 1$, the coordinator sends the request to only one node, C in this case. An old value (0 $) is returned to the client, instead of the expected value (100 $). Strong consistency is not ensured here because $R + W = S$ with the current settings ($S = 3$, $W = 2$, $R = 1$). To achieve strong consistency (i.e., $R + W > S$), we can either:

- increase the number of nodes to read from: on Figure 3.12 the coordinator sends the write operation to all nodes, but again node C does not receive the request. As $W = 2$, the operation is reported as successful to the user. When the user checks the balance, the read operation is sent to

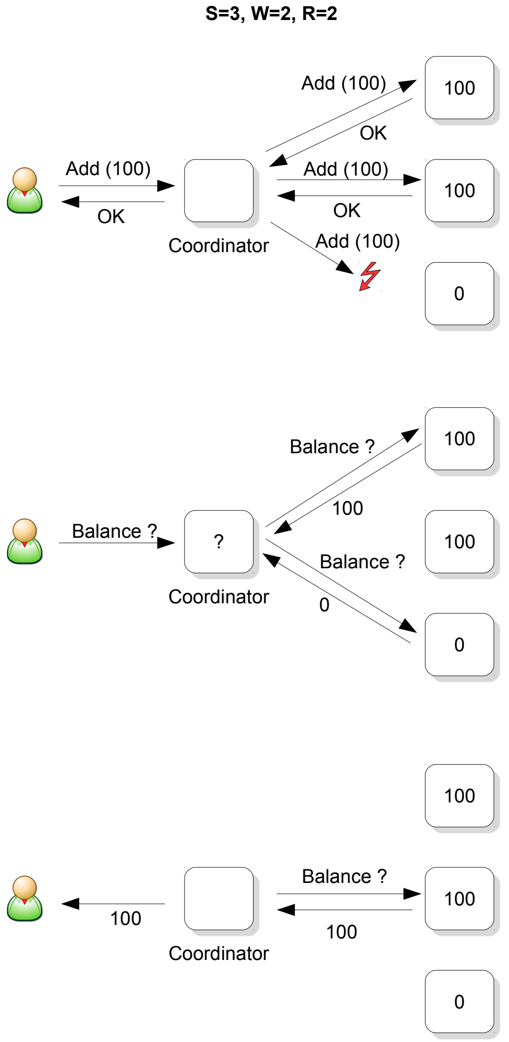


Figure 3.12: Strong Consistency: detecting conflicts at read time

nodes A and C as $R = 2$. The coordinator detects the conflicting values returned by the nodes (0 \$ and 100 \$), and thus asks the third node to form a quorum. The correct value (100 \$) is then returned to the client. An anti-entropy mechanism [81, 90, 129] such as read-repair [78] (see Section 3.5.2) can be used to resynchronize node C.

- increase the number of nodes that need to acknowledge the write operations: on Figure 3.12 as $W = 3$ the write operation fails because node C did not send back its acknowledgement. The transaction is rollbacked and an error is reported to the user.

A large amount of databases ensuring strong consistency adopts $W = S$ and $R = 1$: no need to detect stale values or to resolve conflicts, as all replicas are always synchronized. While the design is simplified, the availability is

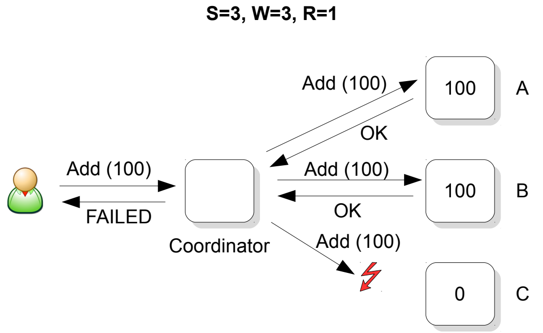


Figure 3.13: Strong Consistency: write failure

reduced, since write operations will fail (or hang) on any replica failure. Another popular setup is $R + W = S + 1$: strong consistency is ensured while achieving greater availability, since replicas can be temporarily unsynchronized.

3.5.2 Eventual Consistency

In an eventual consistent model (i.e., $R + W \leq S$), read operations may return inconsistent values as updates are in progress. After an update, the model only guarantees that subsequent read operations will return the updated value *eventually*. While concurrency control [65, 161] is usually obtained using locking or multiversion concurrency control (MVCC), a database needs to implement several techniques to detect conflicts among unsynchronized replicas and has to replicate data efficiently:

Versioning Detecting conflicts and data versioning are important since some replicas may have a different value for the same key. Vector Clock [85, 61], a type of versioning, is often used to deal with multiple versions of the same value, for example in eventually consistent systems like Dynamo [78] or Voldemort [40]. Another approach used for example by Cassandra [4] is multiversion storage, where a timestamp is stored on each key: when two versions of the same key are in conflict, the version with the most recent timestamp is returned.

Conflict Resolution Several approaches are available to resolve conflicts when they are detected. Dynamo-like systems use a client-side approach where the application is responsible for choosing the correct value or for merging the conflicting versions. A simplified approach taken by Cassandra is to always return the latest value. Hybrid solutions mixing both approaches are also implemented in systems like CouchDB [10].

Hinted Handoff A mechanism to deal with failed nodes is called hinted handoff [78, 4]. If a write operation is sent to a replica that is temporarily unavailable, the system will write a hint to one of the available replicas (or to neighboring nodes or to the coordinator, depending on the systems). This hint suggests that the write operation needs to be replayed to the unavailable node when it has recovered. Hinted handoff provides two advantages: first it reduces the time for a failed node to be synchronized again and second it allows to achieve extreme write availability in the case of consistency is not important.

Anti-Entropy When a node storing the hint for a failed node is down or when a node is unavailable for a long period, the replicas must still synchronize from one-another. A mechanism called anti-entropy [81, 90, 129] compares all the replicas of each key and updates each stale replica to the latest version. Anti-entropy is usually implemented using Merkle Trees [123], which are exchanged by replicas to pinpoint inconsistencies while minimizing the amount of transferred data, as replicas contain mostly similar data.

Read Repair Read repair [78] is an anti-entropy mechanism to optimistically update stale replicas after the coordinator has returned a consistent value from a read request. This mechanism proactively resolves conflicts with little additional work: as every replicas has sent its version of the data to the coordinator, using a conflict-resolution protocol the latter is able to push the latest consistent value to any stale replicas.

Gossiping Gossiping [108, 78] is a classical mechanism to exchange information between a large amount of nodes efficiently and in a scalable manner. A gossip protocol makes sure that eventually every node is aware of the state of the other nodes and is typically used for cluster membership and failure detection.

3.6 Conclusion

Scaling the database layer while ensuring high availability remains one of the most challenging issue of large-scale applications. Different modern approaches try to address the performance and scalability requirements of the database layer. First, the NoSQL databases focus on horizontal scalability in distributed architectures and are designed to address schemaless and non-relational data management. Second, the NewSQL databases offer scalability in distributed architectures while providing relational data management with ACID properties. Third, data cache (or grid) solutions, which are increasingly positioned as potential alternatives to relational databases, are designed to keep data in memory so as to increase database and application performance.

While it is clear that no unique structured storage system fits every class of applications, choosing between relational and non-relational solutions is a non-trivial decision which will have consequences throughout the whole development and life cycle of the application. Distributed database is a hot research topic and few people have a practical experience of such systems. The complexity of maintaining multiple disparate systems and of making them transparent to the application layer requires extra work. The added complexity of distributing the database has also an economic cost in term of infrastructure and engineering: typically, join queries across multiple systems are slow and expensive.

In Chapter 4, we present a highly-available distributed database that replicates data in geographically diverse locations in order to minimize the consequences of correlated failures. Our approach also finds the optimal resource allocation which balances the query processing overhead while being able to handle flash crowds.

Part III

Contributions

Chapter 4

Building Highly-Available and Scalable Cloud Storage

Failures of multiple types are common in current datacenters, partly due to the higher scales of the data stored. As data scales up, supporting its availability becomes more complex, since different availability levels per application or per data item may be required. In this chapter, we propose a self-managed key-value store that dynamically allocates the resources of a data cloud to several applications in a cost-efficient and fair way. Our approach offers and dynamically maintains multiple differentiated availability guarantees to each different application despite failures. We employ a virtual economy, where each data partition (i.e., a key range in a consistent-hashing space) acts as an individual optimizer and chooses whether to migrate, replicate or remove itself based on net benefit maximization regarding the utility offered by the partition and its storage and maintenance cost. As proven by a game-theoretical model, no migrations or replications occur in the system at equilibrium, which is soon reached when the query load and the used storage are stable. Moreover, by means of extensive simulation experiments, we have proven that our approach dynamically finds the optimal resource allocation that balances the query processing overhead and satisfies the availability objectives in a cost-efficient way for different query rates and storage requirements. Finally, we have implemented a fully working prototype of our approach that clearly demonstrates its applicability in real settings.

4.1 Introduction

Cloud storage is becoming a popular business paradigm, e.g., Amazon S3 [2], Google Storage [16], Rackspace Cloud Files [42], etc. Small companies that offer large Web applications can avoid large capital expenditures in infrastructure by renting distributed storage and pay per use. The storage capacity employed may be large and it should be able to further scale up. However, as data scales up, hardware failures in current datacenters become more frequent [130]: e.g., overheating, power (PDU [1]) failures, rack failures, network failures, hard drive failures, network re-wiring and maintenance. Also, geographic proximity significantly affects data availability: e.g., in case of a PDU failure ∼500-1000 machines suddenly disappear, or in case of a rack failure ∼40-80 machines instantly go down. Furthermore, data may be lost due to natural disasters, such as tornadoes destroying a complete data center, or various attacks (DDoS, terrorism, etc.). On the other hand, as [75] suggests, Internet availability varies from 95% to 99.6%. Also, the query rates for Web applications data are highly irregular, e.g. ,the "Slashdot effect" [2], and an application may become temporarily unavailable.

To this end, the support of service level agreements (SLAs) with data availability guarantees in cloud storage is very important. Moreover, in reality, different applications may have different availability requirements. Fault-tolerance is commonly dealt with by replication. Existing works usually rely on randomness to diversify the physical servers that host the data; e.g., in [137], [113] node IDs are randomly chosen, so that peers that are adjacent in the node ID space are geographically diverse with a high probability. To the best of our knowledge, no system explicitly addresses the geographical diversity of the replicas. Also, from the application perspective, geographically distributed cloud resources have to be efficiently utilized to minimize renting costs associated to storage and communication. Clearly, geographical diversity of replica locations and minimizing communication cost are contradictory objectives. From the cloud provider perspective, efficient utilization of cloud resources is necessary both for cost-effectiveness and for accommodating load spikes. Moreover, resource utilization has to be adaptive to resource failures, addition of new resources, load variations and the distribution of client locations.

Distributed key-value store is a widely employed service case of cloud storage. Many Web applications (e.g., Amazon.com) and many large-scale social applications (e.g., LinkedIn, Last.fm, etc.) use distributed key-value stores. Also, several research communities (e.g., peer-to-peer, scalable distributed data structures, databases) study key-value stores, even as complete database solutions (e.g., BerkeleyDB [5]). As a novel contribution of this thesis, we propose a scattered key-value store (referred to as *Skute*), which is designed to provide high and differentiated data availability statistical guarantees to multiple applications in a cost-efficient way in terms of rent price and query response times. Our approach combines the following innovative characteristics:

- It enables a computational economy for cloud storage resources.
- It provides differentiated availability statistical guarantees to different applications despite failures by geographical diversification of replicas.
- It applies a distributed economic model for the cost-efficient self-organization of data replicas in the cloud storage that is adaptive to adding new storage, to node failures and to client locations.

[1] Power Distribution Unit

[2] http://en.wikipedia.org/wiki/Slashdot_effect

- It efficiently and fairly utilizes cloud resources by performing load balancing in the cloud adaptively to the query load.

Optimal replica placement is based on distributed net benefit maximization of query response throughput minus storage as well as communication costs, under the availability constraints. The optimality of the approach is proven by comparing simulation results to those expected by numerically solving an analytical form of the global optimization problem. Also, a game-theoretic model is employed to observe the properties of the approach at *equilibrium*. A series of simulation experiments prove the aforementioned characteristics of the approach. Finally, employing a fully working prototype of *Skute*, we experimentally demonstrate its applicability in real settings.

The rest of the chapter is organized as follows: In Section 4.2, the scattered key-value data store is presented. In Section 4.3, the global optimization problem that we address is formulated. In Section 4.4, we describe the individual optimization algorithm that we employ to solve the problem in a decentralized way. In Section 4.5, we define a game-theoretical model of the proposed mechanism and study its equilibrium properties. In Section 4.6, we discuss the applicability of our approach in an untrustworthy environment. In Section 4.7, we present our simulation results on the effectiveness of the proposed approach. In Section 4.8, we describe the implementation of *Skute* and discuss our experimental results in a real testbed. In Section 4.10, we outline some related work. Finally, in Section 4.11, we conclude our work.

4.2 Skute: Scattered Key-Value Store

Skute is designed to provide low response time on read and write operations, to ensure replicas' geographical dispersion in a cost-efficient way and to offer differentiated availability guarantees per data item to multiple applications, while minimizing bandwidth and storage consumption. The application data owner rents resources from a cloud of federated servers to store its data. The cloud could be a single business, i.e., a company that owns/manages data server resources ("private" clouds), or a broker that represents servers that do not belong to the same businesses ("public" clouds). The number of data replicas and their placement are handled by a distributed optimization algorithm autonomously executed at the servers. Also, data replication is highly adaptive to the distribution of the query load among partitions and to failures of any kind so as to maintain high data availability.

4.2.1 Physical Node

We assume that a physical node (i.e., a server) belongs to a rack, a room, a data center, a country and a continent. Note that finer geographical granularity could also be considered. Each physical node has a label of the form "continent-country-datacenter-room-rack-server" in order to precisely identify its geographical location. For example, a possible label for a server located in a data center in Berlin could be "EU-DE-BE1-C12-R07-S34".

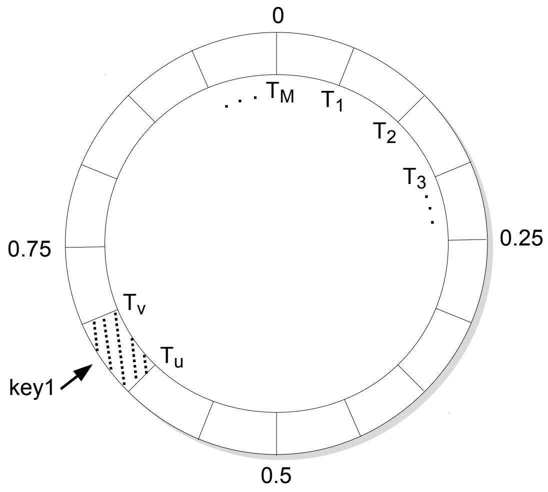


Figure 4.1: Each data item is mapped to a value on a circular range $[0, 1)$, which is split into M equally sized partitions

4.2.2 Virtual Node

In order to scale out, the dataset is partitioned using a variant of consistent hashing [110] and spread over a set of physical nodes to share the load. The output of the consistent hash function is a range $\in [0, 1)$ and is considered as a circular space (i.e., the key space) forming a ring, a flexible and resilient routing geometry [94]. As depicted in Figure 4.1, the key space is split into M equally sized partitions delimited by the positions T_1 to T_M, called tokens. The number of partitions is chosen such that $M >> N$, where N is the number of physical nodes in the system. A data item is identified by a key (produced by a one-way cryptographic hash function such as MD5 [134]), which is used to compute the item position in the ring by (consistent) hashing it. A virtual node (alternatively a replica of a partition) holds data of a partition with a range of keys in $(previous\ token,\ token]$, as in [78]. For example, in Figure 4.1, a data item with the key $key1$ is assigned to the virtual nodes responsible for the key range $(T_u, T_v]$, as $T_u < hash('key1') \leq T_v$. Data of a partition is managed by one or more virtual nodes, where a virtual node may replicate or migrate its data to another server, or suicide (i.e., delete its data replica) according to a decision making process described in Section 4.4.4. A physical node hosts a varying number of virtual nodes depending on the query load, the size of the data managed by the virtual nodes and its own capacity (i.e., CPU, RAM, disk space, etc.). Two virtual nodes managing the same partition are never hosted on the same physical server.

4.2.3 Virtual Ring

Our approach employs the concept of multiple virtual rings on a single cloud in an innovative way (*cf.* Section 4.10 for a comparison with [152]). Thus, as subsequently explained, we allow multiple applications to share the

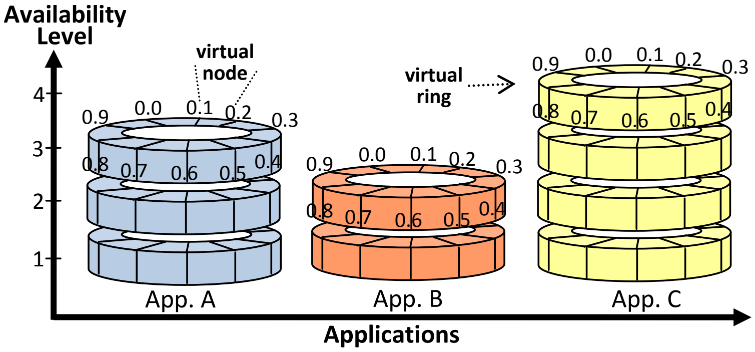


Figure 4.2: Three applications with different availability levels

same cloud infrastructure for offering differentiated per data item and per application availability guarantees without performance conflicts. The single-application case with one uniform availability guarantee has been presented in [66]. In the present work, each application uses its own virtual rings, while one ring per availability level is needed, as depicted in Figure 4.2. Each virtual ring consists of multiple virtual nodes that are responsible for different data partitions of the same application that demand a specific availability level. This approach provides two main advantages over existing key-value stores:

- **Multiple data availability levels per application**: within the same application, some data may be crucial and some may be less important. In other words, an application provider may want to store data with different availability guarantees. Other approaches, such as [78], also argue that they can support several applications by deploying a key-value store per application. However, as data placement for each data store would be independent in [78], an application could severely impact the performance of others that utilize the same resources. Unlike existing approaches, *Skute* allows a fine-grained control of the resources of each server, as every virtual node of each virtual ring acts as an individual optimizer (as described in Section 4.4.4), thus minimizing the impact among applications.

- **Geographical data placement per application**: data that is mostly accessed from a given geographical region should be moved close to that region. Without the concept of virtual rings, if multiple applications were using the same data store, data of different applications would have to be stored in the same partition, thus removing the ability to move data close to the clients. However, by employing multiple virtual rings, *Skute* is able to provide one virtual store per application, allowing the geographical optimization of data placement.

4.2.4 Routing

Skute could be seen as a $O(1)$ distributed hash table (DHT), similarly to [78]. A physical node is responsible to manage the routing table of all virtual rings hosted in it, in order to minimize the update costs. Upon migration, replication and suicide events, hierarchical broadcast that leverages the geographical topology of servers is employed. This approach costs $O(N)$ where N is the number of servers, but it uses the minimum network spanning tree. The position of a moving virtual node (i.e., during the migration process) is tracked by forwarding pointers (e.g., SSP chains [144]). Also, the routing table is periodically updated using a gossiping protocol for shortening/repairing chains or updating stale entries (e.g., due to failures). According to this protocol, a server exchanges with random $log(N)$ other servers the routing entries of the virtual nodes that they are responsible for.

The scalability of this approach is experimentally proven in a real testbed, as described in Section 4.8.

4.2.5 Data Consistency

As a virtual node is allowed to migrate or replicate to a new server, maintaining data consistency during these operations is highly important. We aim to maintain only eventual data consistency among replicas by the use of vector-clock versioning and an anti-entropy mechanism known as read-repair, as in described in Section 3.5.2. In read-repair, conflicting versions of data are presented to the user upon read requests for semantic reconciliation.

For a migration, a virtual node v has to move from a physical node p_1 to another physical node p_2. Once v has elected p_2 as its best candidate for a migration and once p_2 has agreed to host the new virtual node, v broadcasts the routing table update and starts copying its data to p_2. A request for a key belonging to v is now routed to p_2, which knows that this key is currently under migration. In case of a read request, p_2 makes a proxy call to p_1 to return the correct data to the client. A write request is performed locally by p_2, after a proxy read to $p1$ for ensuring proper versioning. Figure 4.3 depicts read and write operations during a migration.

For replication, the virtual node v has to copy itself from a physical node p_1 to another physical node p_2. This process happens similarly to the migration one.

Potential inconsistency among the replicas of v is tolerated and is resolved at read time. As discussed in Section 3.5, three quorum parameters are important:

- R: the number of replicas of v to read from;
- W: the number of replicas of v that acknowledge the write operation before the operation completes;
- S: the number of physical nodes that store replicas of v.

As long as $R + W > S$, strong consistency is ensured because the read set and the write set always overlap on at least one element. When $R + W \leq S$,

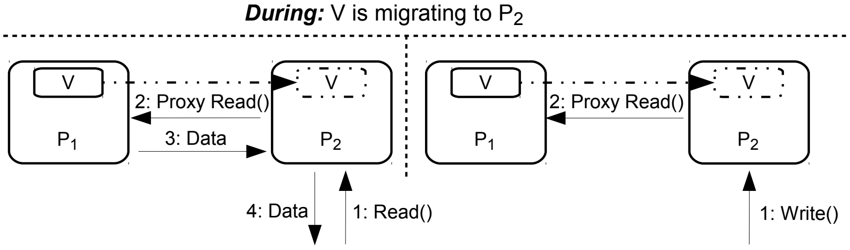


Figure 4.3: Data consistency during a virtual node migration

the read set and the write sets might not overlap, hence only weak/eventual consistency (WEC) can be guaranteed. Configuring the values of R, W and S is a tradeoff between performance and consistency as shown in Table 4.1, and depends on the application needs.

Table 4.1: Example of quorum parameters *(SC: strong consistency, WEC: Weak/Eventual consistency)*

R	W	S	Comment
2	2	3	SC, focus on fault tolerance
1	3	3	SC, focus on consistency
1	2	3	WEC only
1	6	10	WEC only, focus on very high read load

Contrary to [78], the number of replicas, S, is changing dynamically in case of replication or suicide. Recall that S is stored at the routing table of the physical node. The values for R and W are selected as functions of S on a per virtual ring basis based on the performance needs.

4.3 Problem Definition

The data belonging to an application is split into M partitions, where each partition i has r_i distributed replicas. We assume that N servers are present in the data cloud.

4.3.1 Maximize Data Availability

The first objective of a data owner d (i.e., application provider) is to provide the highest availability for a partition i, by placing all of its replicas in a set S_i^d of different servers. Data availability generally increases with the geographical diversity of the selected servers. Obviously, the worst solution in terms of data availability would be to put all replicas at a server with equal or worse probability of failure than others.

We denote as F_j a failure event at server $j \in S_i^d$. These events may be independent from each other or correlated. If we assume without loss of generality that events $F_1 \dots F_k$ are independent and that events $F_{k+1} \dots F_{|S_i^d|}$ are correlated, then the probability a partition i to be unavailable is given as follows:

$$
\begin{aligned}
& Pr(i \text{ unavailable}) = Pr(F_1 \cap F_2 \cap \dots \cap F_{|S_i^d|}) = \\
& \prod_{j=1}^{k} Pr(F_j) \ \cdot \ Pr(F_k | F_{k+1} \dots \cap F_{|S_i^d|}) \cdot \\
& Pr(F_{k+1} | F_{k+2} \cap \dots \cap F_{|S_i^d|}) \cdot \dots \cdot Pr(F_{|S_i^d|}) \,,
\end{aligned}
\tag{4.1}
$$

if $F_{k+1} \cap F_{k+2} \cap \dots F_{|S_i^d|} \neq \oslash$.

Otherwise, when $F_{k+1} \cap F_{k+2} \cap \dots F_{|S_i^d|} = \oslash$, the events are uncorrelated and $Pr(F_{k+1} | F_{k+2} \cap \dots \cap F_{|S_i^d|}) \cdot \dots \cdot Pr(F_{|S_i^d|}) = 1$.

4.3.2 Minimize Communication Cost

While geographical diversity increases availability, it is also important to take into account communication cost among servers that host different replicas, in order to save bandwidth during replication or migration, and to reduce latency in data accesses and during conflict resolution for maintaining data consistency. Let $\overset{\rightrightarrows}{L^d}$ be a $M \times N$ *location* matrix with its element $L_{ij}^d = 1$ if a replica of partition i of application d is stored at server j and $L_{ij}^d = 0$ otherwise. Then, we maximize data proximity by minimizing network costs for each partition i, e.g., the total communication cost for conflict resolution of replicas for the mesh network of servers where the replicas of the partition i are stored. In this case, the network cost c_n for conflict resolution of the replicas of a partition i of application d can be given by

$$
c_n(\vec{L_i^d}) = sum(\vec{L_i^d} \cdot \overset{\rightrightarrows}{NC} \cdot \vec{L_i^d}^T) \,,
\tag{4.2}
$$

where $\overset{\rightrightarrows}{NC}$ is a strictly upper triangular $N \times N$ matrix whose element NC_{jk} is the communication cost between servers j and k, and sum denotes the sum of matrix elements.

4.3.3 Maximize Net Benefit

Every application provider has to periodically pay the operational cost of each server where he stores replicas of his data partitions. The operational cost of a server is mainly influenced by the quality of the hardware, the physical hosting, the access bandwidth and storage allocated to the server, and the query processing and communication overhead. The data owner wants to minimize his expenses by replacing expensive servers with cheaper ones, while maintaining a minimum data availability promised by SLAs to his clients. He also obtains some utility $u(.)$ from the queries answered by its data replicas that depends on the popularity (i.e., query load) pop_i of the data contained in the replica of the partition i and the response time (i.e., processing and network latency) associated to the replies. The network latency depends on the distance of the clients from the server that hosts the data, i.e., the geographical distribution G_i of query clients. Overall, he seeks to maximize his net benefit and the global optimization problem can be formulated as follows:

$$\begin{aligned} &\max\{u(pop_i, G_i) - \vec{L_i^d}\ \vec{c}^T + c_n(\vec{L_i^d})\}, \forall i,\ \forall d \\ &\text{s.t.} \\ &1 - Pr(F_1^{L_{i1}^d} \cap F_2^{L_{i2}^d} \cap \ldots F_N^{L_{iN}^d}) \geq th_d\,, \end{aligned} \tag{4.3}$$

where $\vec{c}$ is the vector of operational costs of servers with its element c_j being an increasing function of the data replicas of the various applications located at server j. This also accounts for the fact that the processing latency of a server is expected to increase with the occupancy of its resources. F_j^0 for a particular partition denotes that the partition is not present at server j and thus the corresponding failure event at this server is excluded from the intersection and th_d is a minimum availability threshold promised by the application provider d to clients. This constrained global optimization problem takes $2^{M \cdot N}$ possible solutions for every application and its solution is computationally tractable only for small sets of servers and partitions.

4.4 The Individual Optimization

The data owner rents storage space located in several data centers around the world and pays a monthly usage-based *real rent*. Each virtual node is responsible for the data in its key range and should always try to keep data availability above a certain minimum level required by the application while minimizing the associated costs (i.e., for data hosting and maintenance). To this end, a virtual node can be assumed to act as an autonomous agent on behalf of the data owner to achieve these goals. Time is assumed to be split into epochs. A virtual node may replicate or migrate its data to another server, or suicide (i.e., delete its data replica) at each epoch and pay a *virtual rent* (i.e., an approximation of the possible real rent, defined later in this section) to servers where its data are stored. These decisions are made based on the query rate for the data of the virtual node, the renting costs and the maintenance of high availability upon failures. There is no global coordination and each virtual node behaves independently. Only one virtual node of the same partition is allowed to suicide at the same epoch by employing Paxos [114] distributed consensus algorithm among virtual nodes of the same partition. The virtual rent of each server is announced at a board and is updated at the beginning of a new epoch.

4.4.1 Board

At each epoch, the virtual nodes need to know the virtual rent price of the servers. Each server maintains its own local board, a list linking each server with its virtual rent price, and periodically updates the virtual prices of a random subset ($log(N)$) of servers by contacting them directly (i.e., gossiping), having as N the total number of servers. This fully decentralized architecture has been experimentally verified in a real testbed to be very efficient without creating high communication overhead (*cf.* Section 4.8).

When a new server is added to the network, the data owner estimates its *confidence level* based on its hardware components and its location. This estimation depends on technical factors (e.g., redundancy, security, etc.) as well as non-technicals ones (e.g., political and economical stability of the country, etc.) and it is rather subjective.

4.4.2 Physical Node

The virtual rent price c of a physical node for the next epoch is an increasing function of its query load and its storage usage at the current epoch and it can be given by:

$$c = up \cdot (1 + \alpha \cdot storage\_usage + \beta \cdot query\_load)\,, \tag{4.4}$$

where α, β are normalizing factors and up is the unit price for marginal usage of the server, which can be calculated by dividing the real rent of the previous billing period (e.g., an hour) to the mean usage of the server in the previous billing period. We consider that the real rent price per server takes into account the network cost for communicating with the server, i.e., its access link. To this end, it is assumed that access links are going to be the bottleneck ones along the path that connects any pair of servers and thus we do not take explicitly into account distance between servers. Multiplying the real rent price with the query load satisfies the network proximity objective. The query load and the storage usage at the current epoch are considered to be good approximations of the ones at the next epoch, as they are not expected to change very often at very small time scales, such as a time epoch. The virtual rent price per epoch is an approximation of the real monthly price that is paid by the application provider for storing the data of a virtual node. Thus, an expensive server tends to be also expensive in the virtual economy. A server agent residing at the server calculates its virtual rent price per epoch and updates the board.

4.4.3 Maintaining Availability

A virtual node always tries to keep the data availability above a minimum level th_d (i.e., the availability level of the corresponding virtual ring), as specified in Section 4.3. As estimating the probabilities of each server to fail necessitates access to a large set of historical data and private information of the server, we approximate the potential availability of a partition (i.e., virtual node) by means of the geographical diversity of the servers that host its replicas. Therefore, the availability of a partition i is defined as the sum of *diversity* of each distinct pair of servers, i.e.,:

$$avail_i = \sum_{k=0}^{|S_i|} \sum_{j=k+1}^{|S_i|} conf_k \cdot conf_j \cdot diversity(s_k, s_j) \tag{4.5}$$

where $S_i = (s_1, s_2, \ldots, s_n)$ is the set of servers hosting replicas of the virtual node i and $conf_k$, $conf_j \in [0,\ 1]$ are the confidence levels of servers k, j. The diversity function returns a value calculated based on the geographical distance among each server pairs. This distance is represented as a 6 bit number, each bit corresponding to the location parts of a server, namely continent, country, data center, room rack and server. Note that more bits would be required to represent additional geographical location parts than those considered. The most significant bit (leftmost) represents the continent while the least significant bit (rightmost) represents the server. Starting with the most significant bit, each location part of both servers are compared one by one to compute their *similarity*: if the location parts are equivalent, the corresponding bit is set to 1, otherwise 0. Once a bit has been set to 0, all less significant bits are also set to 0. For example, two servers belonging to the same data center but located in different rooms cannot be in the same rack, thereby all bits after the third bit (data center) have to be 0. The similarity number would then look like this:

cont	coun	data	room	rack	serv
1	1	1	0	0	0

A binary "NOT" operation is then applied to the similarity to get the diversity value:

$$\overline{111000} = 000111 = 7(decimal)$$

The diversity values of server pairs are summed up, because having more replicas in distinct servers located even in the same location always results in increased availability.

When the availability of a virtual node falls below th, it replicates its data to a new server. Note that a virtual node can know the locations of the replicas of its partition from the routing table of its hosting server and thus calculate its availability according to 4.5. The best candidate server is selected so as to maximize the net benefit between the diversity of the resulting set of replica locations for the virtual node and the virtual rent of the new server. Also, a preference weight g_j is associated to a server j according to its location proximity to the geographical distribution G_i of clients querying a virtual node managing the partition i. G_i is approximated by the virtual node by storing the number of client queries per location l. Thus, the availability is increased as much as possible at the minimum cost, while the network latency for the query reply is decreased. Specifically, a virtual node managing a partition i with current replica locations in S_i maximizes:

$$\arg\max_j \sum_{k=0}^{|S_i|} (g_j \cdot conf_j \cdot diversity(s_k, s_j) - c_j)\,, \tag{4.6}$$

where c_j is the virtual rent price of candidate server j. g_j is the weight related to the proximity (i.e., inverse average diversity) of the location of the candidate server j to the geographical distribution of clients querying the partition i managed by the virtual node and is given by:

$$g_j = \frac{\sum_l q_l}{1 + \sum_l q_l \cdot diversity(l, s_j)}, \tag{4.7}$$

where q_l is the number of queries for the partition i of the virtual node per client location l. To this end, we assume that the client locations are encoded similarly to those of servers. In fact, if client requests reach the cloud by the geographically nearest cloud node to the client (e.g., by employing geoDNS), we can take the location of this cloud node as the client location. However, having virtual nodes to choose the destination server j for replication according to (4.6) would render j a bottleneck for the next epoch. Instead, the destination server is randomly chosen among the top-k (with $k = \lceil ln(N) \rceil$) ones, ranked according to the maximized quantity in (4.6).

The minimum availability level th allows a fine-grained control over the replication process. A low value means that a partition will be replicated on few servers potentially geographically close, whereas a higher value enforces many replicas to be located at diverse locations. However, setting a high value for the minimum level of availability in a network with a few servers can result in an undesirable situation, where all partitions are replicated everywhere. To circumvent this, a maximum number of replicas per virtual node is allowed.

4.4.4 Virtual Node Decision Tree

As already mentioned, a virtual node agent may decide to replicate, migrate, suicide or do nothing with its data at the end of an epoch. Note that decision making of virtual nodes does not need to be synchronized. Upon replication, a new virtual node is associated with the replicated data. The decision tree of a virtual node is depicted in Figure 4.4. First, it verifies that the current availability of its partition is greater than th. If the minimum acceptable availability is not reached, the virtual node replicates its data to the server that maximizes availability at the minimum cost, as described in Subsection 4.4.3.

If the availability is satisfactory, the virtual node agent tries to minimize costs. During an epoch, virtual nodes receive queries, process them and send the replies back to the client. Each query creates a *utility value* for the virtual node, which can be assumed to be proportional to the size of the query reply and inversely proportional to the average geographical distance of the client locations from the server of the virtual node. For this purpose, the balance (i.e., net benefit) b for a virtual node managing a partition i is defined as follows:

$$b = u(pop, G_i) - c, \tag{4.8}$$

where $u(pop, G_i) = \frac{f(pop)}{h(G_i)}$ is assumed to be the epoch query load of the partition with a certain popularity pop divided by the average proximity of the virtual node to the client locations and normalized to monetary units, and c is the virtual rent price. $f()$ and $h()$ are two increasing functions. To this end, a virtual node decides to:

I) Migrate or Suicide If it has negative balance for the last f epochs, then it migrates or suicides. First, the virtual node calculates the availability of its partition without its own replica. If the availability is satisfactory, the virtual node suicides, i.e., deletes its replica. Otherwise, the virtual node tries to find a less expensive (i.e., busy) server that is closer to the client locations (according to maximization formula (4.6)). To avoid a data replica oscillation among servers, the migration is only allowed if the following *migration conditions* apply:

- The minimum availability is still satisfied using the new server,
- the absolute price difference between the current and the new server is greater than a threshold,
- the current server storage usage is above a storage soft limit, typically 70% of the hard drive capacity, and the new server is below that limit.

II) Replicate If it has positive balance for the last f epochs, it may replicate. For replication, a virtual node has also to verify that:

- It can afford the replication by having a positive balance b' for consecutive f epochs:

  $$b' = u(pop, G_i) - c_n - (1 + \varphi) \cdot c'$$

  where c_n is a term accounting for the network cost to maintain consistency between the replicas of the virtual nodes managing the same partition, which can be approximated as the number of replicas of the partition times a fixed average communication cost parameter for conflict resolution and routing table maintenance. c' is the current virtual rent of the candidate server for replication (randomly selected among the top-k ones ranked according to the formula (4.6)), while the factor (1+φ) accounts for a $\varphi \cdot 100\%$ increase at this rent price at the next epoch due to the potentially increased occupied storage and query load of the candidate server (an upper bound of $\varphi = 0.2$ can typically be assumed). This action aims to distribute the load of the current server towards one located closer to the clients. Thus, it tends to decrease the processing and network latency of the queries for the partition.

- the average bandwidth consumption bdw_r for answering queries per replica after replication (left term of left side of inequality (4.9)) plus the bandwidth used to replicate the partition of size p_s is less than the respective bandwidth bdw per replica without replication (right side of inequality (4.9)) for a fixed number win of epochs to compensate for steep changes of the query rate. A large win value should be used for bursty query load. Specifically, the virtual node replicates if:

  $$\frac{win * q * q_s}{|S_i| + 1} + p_s < \frac{win * q * q_s}{|S_i|}, \tag{4.9}$$

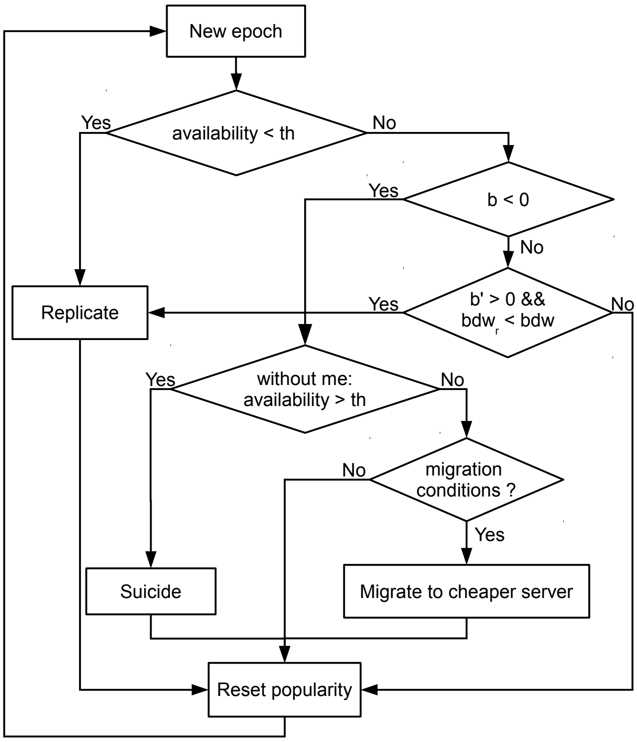


Figure 4.4: Decision tree of the virtual node agent

where q is the average number of queries for the last win epochs, q_s is the average size of the replies, $|S_i|$ is the number of servers currently hosting replicas of partition i and p_s is the size of the partition.

At the end of a time epoch, the virtual node agent sets the lowest utility value $u(pop, G_i)$ to the current lowest virtual rent price among the servers to prevent unpopular nodes from migrating indefinitely. Otherwise, unpopular virtual nodes would have a negative balance at every epoch, forcing them to migrate to cheaper servers. As the decision process is not centralized nor synchronized between the virtual nodes, the unpopular virtual nodes would continuously be able to find servers with cheaper virtual rent prices, mainly because of the effect on virtual rent prices of migrations of other unpopular virtual nodes.

A virtual node that either gets a large number of queries or has to provide large query replies becomes wealthier. At the same time, the load of its hosting server will increase, as well as the virtual rent price for the next epoch. Popular virtual nodes on the server will have enough "money" to pay the growing rent price, as opposed to unpopular ones that will have to move to a cheaper server. The transfer of unpopular virtual nodes will in turn decrease the virtual rent price, hence stabilizing the rent price of the server. This approach is self-adaptive and balances the query load by replicating popular virtual nodes.

4.5 Equilibrium Analysis

Without global coordination, we have to ensure that the system will eventually reach equilibrium under stable conditions. As virtual nodes are able to migrate, replicate, stay or suicide, their behaviour have to be analysed in order to make sure that no virtual node will keep migrating indefinitely. Assume that M is the original number of partitions (i.e., virtual nodes) in the system. These virtual nodes may belong to the same or to different applications and compete among each other. Time is assumed to be slotted in rounds. At each round, a virtual node (which is either responsible for an original partition or a replica of the partition) is able to migrate, replicate, suicide (i.e., delete itself) or do nothing (i.e., stay) competitively to other virtual nodes in a repeated game. Before expressing the expected payoffs (i.e., a number reflecting the desirability of an outcome) of the actions of a virtual node, let us define several quantities:

- $u_i^{(t)}$ is the utility gained by the queries served by the partition for which virtual node i is responsible and only depends on its popularity at round t; for simplicity and without loss of generality, we assume that clients are geographically uniformly distributed. To this end, a virtual node expects that this popularity will be maintained at the next round of the game.
- $r_i^{(t)}$ is the number of replicas of virtual node i at round t.
- $C_c^{(t+1)}$ is the expected price at round $t+1$ of the cheapest server at round t. Note that it is a dominant strategy for each virtual node to select the cheapest server to migrate or replicate to, as any other choice could be exploited by competitive rational virtual nodes.
- $C_e^{(t+1)}$ is the price at round $t+1$ of the current hosting server at round t, therefore $C_e^{(t)} > C_c^{(t)}$. In case of replication, two virtual nodes will henceforth exist in the system, having equal expected utilities, but the old one paying $C_e^{(t+1)}$ and the new one paying $C_c^{(t+1)}$.
- f_d^i is the mean communication cost per replica of virtual node i for data consistency.
- f_c^i is the communication cost for migrating virtual node i.
- $a_i^{(t)}$ is the utility gain due to the increased availability of virtual node i when a new replica is created.

The expected single round strategy payoffs at round $t+1$ by the various actions made at round t of the game for a virtual node i are given by:

- *Migrate:* $EV_M = \frac{u_i^{(t)}}{r_i^{(t)}} - f_c^i - f_d^i \cdot r_i^{(t)} - C_c^{(t+1)}$
- *Replicate:*
  $EV_R = \frac{u_i^{(t)}+a_i^{(t)}}{r_i^{(t)}+1} - f_c^i - f_d^i(r_i^{(t)}+1) - \frac{1}{2}(C_c^{(t+1)} + C_e^{(t+1)})$

- *Suicide:* $EV_D = 0$

- *Stay:* $EV_S = \frac{u_i^{(t)}}{r_i^{(t)}} - f_d^i r_i^{(t)} - C_e^{(t+1)}$

In the aforementioned formula of EV_R, we calculate the expected payoff per copy of the virtual node after replication. Notice that EV_R is expected to be initially significantly above 0, as the initial utility gain a from availability should be large in order the necessary replicas to be created. Also, if the virtual price difference among servers is initially significant, then $EV_M - EV_S$ will be frequently positive and virtual nodes will migrate towards cheaper servers. As the number of replicas increases, a decreases (and eventually becomes a small constant close to 0 after the required availability is reached) and thus EV_R decreases. Also, as price difference in the system is gradually balanced, the difference $EV_M - EV_S$ becomes more frequently negative, so fewer migrations happen. On the other hand, if the popularity (i.e., query load) of a virtual node is significantly deteriorated (i.e., u decreases), while its minimum availability is satisfied (i.e., a is close to 0), then it may become preferable for a virtual node to commit suicide.

Next, we consider the system at equilibrium and that the system conditions, namely the popularity of virtual nodes and the number of servers, remain stable. If we assume that each virtual node i plays a mixed strategy among its pure strategies, specifically it plays migrate, replicate, suicide and stay with probabilities x, y, z and $1 - x - y - z$ respectively, then we calculate $C_c^{(t+1)}$, $C_e^{(t+1)}$ as follows:

$$C_c^{(t+1)} = C_c^{(t)}[1 + (x + y) \textstyle\sum_{i=1}^{M} r_i^{(t)}] \tag{4.10}$$

$$C_e^{(t+1)} = C_e^{(t)}[1 - (x + z + \varphi y)] \tag{4.11}$$

In equation (4.10), we assume that the price of the cheapest server at the next time slot increases linearly with the number of replicas that are expected to have migrated or replicated to that server until the next time slot. Also, in equation (4.11), we assume that the expected price of the current server at the next time slot decreases linearly with the fraction of replicas that are expected to abandon this server or replicate until the next time slot. $0 < \varphi << 1$ is explained as follows: recall that the total number of queries for a partition is divided by the total number of replicas of that partition and thus replication also reduces the rent price of the current server. However, the storage cost for hosting the virtual node remains and, as the number of replicas of the virtual node in the system increases, it becomes the dominant cost factor of the rent price of the current server. Therefore, replication only contributes to $C_e^{(t+1)}$ in a limited way, as shown in equation (4.11). Note that any cost function (e.g., a convex one, as storage is a constrained resource) could be used in our equilibrium analysis, as long as it was increasing to the number of replicas, which is a safe assumption.

Henceforth, for simplicity, we drop i indices as we deal only with one virtual node. Recall that the term a becomes close to 0 at equilibrium. Then, the replicate strategy is dominated by the migrate one, and thus $y = 0$. Also, the suicide strategy has to be eventually dominated by the migrate and stay strategies, because otherwise every virtual node would have have the incentive to leave the system; thus $z = 0$. Therefore, the number r of replicas of a virtual node becomes fixed at equilibrium and the total sum N_r of the replicas of all virtual nodes in the cloud is also fixed. As $y = z = 0$ at equilibrium, the virtual node plays a mixed strategy among migrate and stay with probabilities x and $1 - x$ respectively. The expected payoffs of these strategies should be equal at equilibrium, as the virtual node should be indifferent between them:

$$\begin{aligned}
&EV_M = EV_S \Leftrightarrow \\
&\frac{u}{r} - f_c - f_d\, r - C_c(1 + x\, N_r) = \frac{u}{r} - f_d\, r - C_e(1 - x) \Leftrightarrow \\
&x = \frac{C_e - C_c - f_c}{C_e + C_c\, N_r}
\end{aligned} \tag{4.12}$$

The nominator of x says that in order for any migrations to happen in the system at equilibrium the rent of the current server used by a virtual node should exceed the rent of the cheapest server more than the cost of migration for this virtual node. Also, the probability to migrate decreases with the total number of replicas in the system. Considering that each migration decreases the average virtual price difference in the system, then the number of migrations at equilibrium will be almost 0.

4.6 Rational Strategies

We have already accounted for the case that virtual nodes are rational, as we have considered them to be individual optimizers. In this section, we consider the rational strategies that could be employed by servers in an untrustworthy environment. No malicious strategies are considered, such as tampering with data, deliberate data destruction or theft, because standard cryptographic methods (e.g., digital signatures, digital hashes, symmetric encryption keys) could easily alleviate them (at a performance cost) and the servers would have legal consequences if discovered employing them. Such cryptographic methods should be employed in a real untrustworthy environment, but we refrain from further dealing with them in this chapter. However, rational servers could have the incentive to lie about their virtual prices, so that they do not reflect the actual usage of their storage and bandwidth resources. For example, a server may over-utilize its bandwidth resources by advertising a lower virtual price (or equivalently a lower bandwidth utilization) than the true one and increase its profits by being paid by more virtual nodes. At this point, recall that the application provider pays a monthly rent per virtual node to each server that hosts its virtual nodes. In case of server over-utilization, some queries to the virtual nodes of the server would have to be buffered or even dropped by the server. Also, one may argue that a server can increase its marginal usage price at will in this environment, which then is used to calculate the monthly rent of a virtual node. This is partly true, despite competition among servers, as the total actual resource usage of a server per month cannot be easily estimated by individual application providers.

The aforementioned rational strategies could be countered as follows: in Section 4.4, we assumed that virtual nodes assign to servers a subjective confidence value based on the quality of the resources of the servers and their location. In an untrustworthy environment, the confidence value of a server could also reflect its *trustworthiness* for reporting its utilization correctly. This trustworthiness value could be effectively approximated by the application provider by means of reputation based on periodical monitoring of the performance of servers to own queries. The aforementioned rational strategies are common in everyday transactions among sellers and buyers, but in a competitive environment, comparing servers based on their prices and their offered performance provides them with the right incentives for truthful reporting [80]. Therefore, in a cloud with rational servers, application providers should divide the virtual rent price c_j by the confidence $conf_j$ of the server j in the maximization formula (4.6), in order to provide incentives to servers to refrain from employing the aforementioned strategies.

4.7 Simulation Results

4.7.1 The Simulation Model

We assume for our simulation a cloud storage environment consisting of N servers geographically distributed according to different scenarios that are explained on a per case basis. Data per application is assumed to be split into M partitions having each represented by a virtual node. Each server has fixed bandwidth capacities for replication and migration per epoch. They also have a fixed bandwidth capacity for serving queries and a fixed storage capacity. All servers are assumed to be assigned the same confidence. The popularity of the virtual nodes (i.e., the query rate) is distributed according to the Pareto distribution, which has the following probability density function:

$$f_X(x) = \frac{\alpha * x_m^\alpha}{x^{\alpha+1}} \qquad for\, x \geq x_m \tag{4.13}$$

where x_m is the minimum possible value of a random variable X, and α is the shape parameter. In our simulation, we set $x_m = 1$ and $\alpha = 50$ as parameters of the distribution, which will be referred to as Pareto(1, 50) in the remainder of this thesis. The number of queries per epoch is Poisson distributed with a mean rate λ, which is different per experiment. For facilitating the comparison of the simulation results with those of the analytical model of Section 4.3, the geographical distribution of query clients is assumed to be uniform and thus g_j is 1 for any server j. The size of every data partition is assumed to be fixed and equal to 256MB. Time is considered to be slotted into epochs. At each epoch, virtual nodes employ the decision making algorithm of Subsection 4.4.4. Note that decision making of virtual nodes is not synchronized. Each server updates its available bandwidth for migration, replication or answering queries, and its available storage after every data transfer that is decided to happen within one epoch. Previous data migrations and replications are taken into account in the next epoch. The virtual price per server is determined according to formula (4.4) at the beginning of each epoch.

4.7.2 Convergence to Equilibrium and Optimal Solution

We first consider a *small scale* scenario to validate our results solving numerically the optimization problem of Section 4.3. Specifically, we consider a data cloud consisting of $N = 5$ servers dispersed in Europe: two servers are hosted in Switzerland in separate data centers, one in France and two servers are hosted in Germany in the same rack of the same data center. Data belongs to two applications and it is split into $M = 50$ partitions per application that are randomly shared among servers at startup. The mean query rate is $\lambda = 300$ queries per epoch. The minimum availability level in the simulation model is configured so as to ensure that each partition of the first (resp. second) application is hosted by at least 2 (resp. 4) servers located at different data centers. In the analytical model of Section 4.3, we assume a bad scenario where each server has probability 0.3 to fail and that the failure probabilities of the first 3 server are independent, while those of the Germany data centers are correlated, so as $Pr[F_4|F_5] = Pr[F_5|F_4] = 0.5$. We set $th_1 = 0.9$ for the first application and $th_2 = 0.985$ for the second application. Only network-related operational costs (i.e., access links) are considered the dominant factor for the communication cost and thus distance of servers is not taken into account in decision making; therefore we assume $c_n = 0$, in both the simulation and the analytical model. The same confidence is assigned to all servers in the simulation model. The monthly operational cost c of each server is assumed to be 100$. Also, as the geographical distribution of query clients is assumed to be uniform, the utility function in the analytical model only depends on the popularity pop_i of the virtual node i and is taken equal to $100 \cdot pop_i$. The detailed parameters of this experiment are shown in the left column (small scale) of Table 4.2.

As depicted in Figure 4.5, the virtual nodes start replicating and migrating to other servers and the system soon reaches *equilibrium*, as predicted in Section 4.5. The convergence process actually takes only about 8 epochs, which is very close to the communication bound for replication (i.e., total data size / replication bandwidth = 10GB / 1.5GB per epoch≈ 6.6 epochs). Also, as revealed by comparing the numerical solution of the optimization problem of Section 4.3 with the one that is given by simulation experiments, the proposed distributed economic approach solves rather accurately the optimization problem. Specifically, the average number of virtual nodes of either application per server were the same and the distributions of virtual nodes of either application per server were similar.

4.7.3 Fault Tolerance against Correlated Failures and Adaptation to New Resources

Failures of servers within a cloud are correlated: if a rack fails, every server in it will also fail. The same holds of course for rooms or even datacenters.

Table 4.2: Parameters of small-scale and large-scale experiments.

Parameter	Small scale	Large scale
Servers	5	200
Server storage	10 GB	10 GB
Server price	100$	100$ (70%), 125$ (30%)
Total data	10 GB	100 GB
Average size of an item	500 KB	500 KB
Partitions	50	10000
Queries per epoch	Poisson (λ = 300)	Poisson (λ = 3000)
Query key distribution	Pareto (1,50)	Pareto (1,50)
Storage soft limit	0.7	0.7
Win	20	100
Replication bandwidth	300 MB/epoch	300 MB/epoch
Migration bandwidth	100 MB/epoch	100 MB/epoch

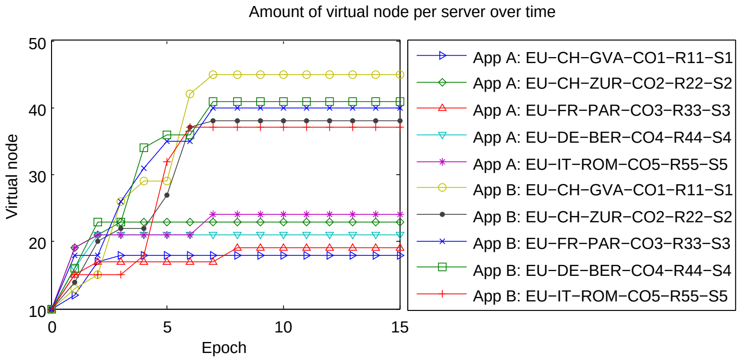


Figure 4.5: Small-scale scenario: replication process at startup

In this experiment, we compare the fault-tolerance of our geographical placement strategy with that of random replica placement, which is a commonly used strategy. We assume 18 racks, each containing 10 servers that are physically hosted as depicted in the bottom-right side of Figure 4.6. We simulate concurrent rack failures in the cases of 2, 3 and 4 replicas per data item, by concurrently shutting down 1 up to 18 racks (all racks). Each experiment is repeated 5 times. Figure 4.6 shows the percentage of data lost due to the rack failures. Clearly, our geographical replica placement outperforms the random placement strategy. For example, in the case of 2 replicas per data item, the geographical placement can always sustain a datacenter failure (i.e., racks 1 to 6 concurrently crash) without any data loss, as opposed to the random placement strategy where more than 10% of the data is lost.

Also, we consider a *large-scale* scenario in which data belongs to three different applications with a requirement for a minimum number of 2, 3, 4 replicas

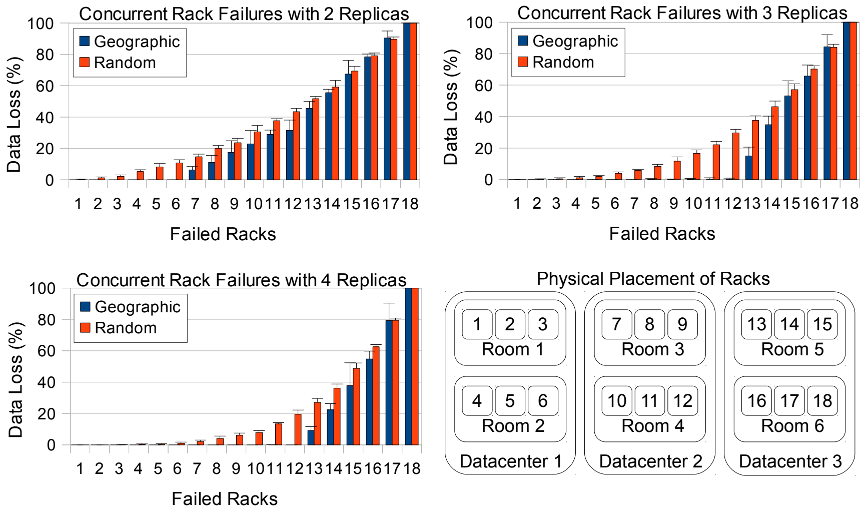


Figure 4.6: Concurrent rack failures with 2 replicas (*top*, *left*) and 3 replicas (*top*, *right*). Concurrent rack failures with 4 replicas (*bottom*, *left*); Physical placement of racks within 3 datacenters (*bottom*, *right*)

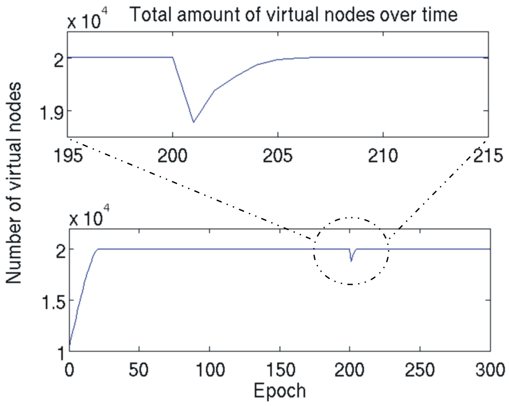


Figure 4.7: Large-scale scenario: robustness against upgrades and failures. The *top* figure only depicts the 20 epochs close to the removal (epoch 200) of 30 randomly chosen servers, while the *bottom* figure depicts the complete experiment with both the addition (epoch 100) and the removal of servers.

respectively. Servers are shared among 10 countries with 2 datacenters per country, 1 room per datacenter, 2 racks per room, and 5 servers per rack. The other parameters of this experiment are shown in the right column (large-scale) of Table 4.2. At epoch 100, we assume that 30 new servers are added to the data cloud, while 30 random servers are removed at epoch 200. As depicted in Figure 4.7, our approach is very robust to resource upgrading or failures: the total number of virtual nodes remains constant after adding resources to the data cloud and increases upon failure to maintain high availability.

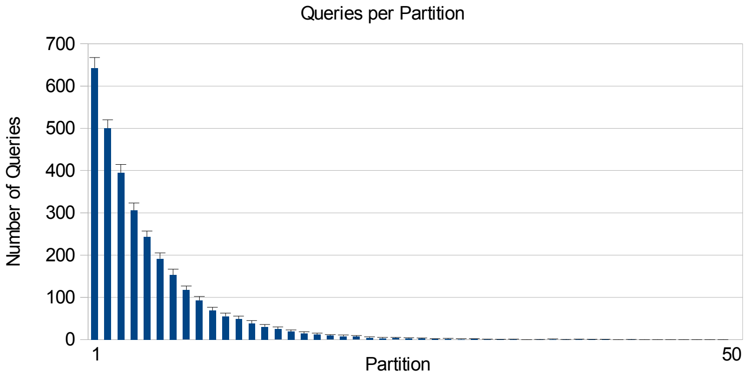


Figure 4.8: Number of Queries per Partition

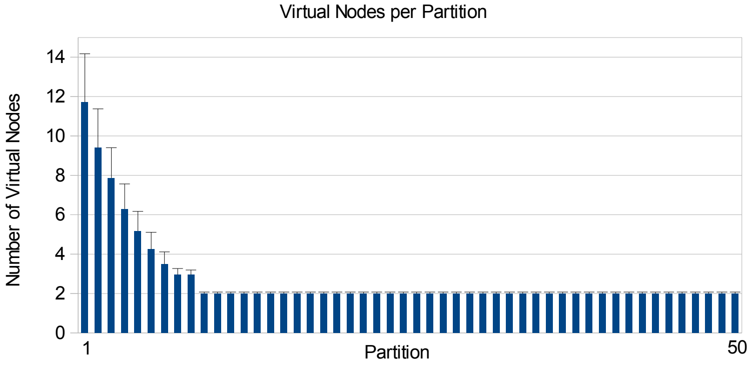


Figure 4.9: Number of Virtual Nodes per Partition

4.7.4 Adaptation to the Query Load

In order the system to be adaptive, the number of replicas of a partition should depend on the query load to that partition. In the first simple scenario, a single virtual ring composed of $M = 50$ partitions is considered. The minimum number of replicas per partition is two and the number of servers is $N = 20$. A constant rate of 3000 queries per epoch following the Pareto(1, 50) distribution are sent to the key-value store. As depicted in Figure 4.8, a few number of partitions hosting popular keys receive most of the query load. The overloaded virtual nodes react and replicate, such that the number of virtual nodes of a partition corresponds to the popularity of the keys belonging to a partition, as depicted in Figure 4.9.

Next, we simulate a load peak similar to what would result from the Slashdot effect [3]: in a short period the query rate gets multiplied by 60. Hence, at epoch 200 the mean rate of queries per epoch increases from 3000 to 183000 in 25 epochs and then slowly decreases for 250 epochs until it reaches the normal rate of 3000 queries per epoch. Following the *large-scale* scenario of Table 4.2, the data belongs to three different applications with a requirement

[3] http://en.wikipedia.org/wiki/Slashdot_effect

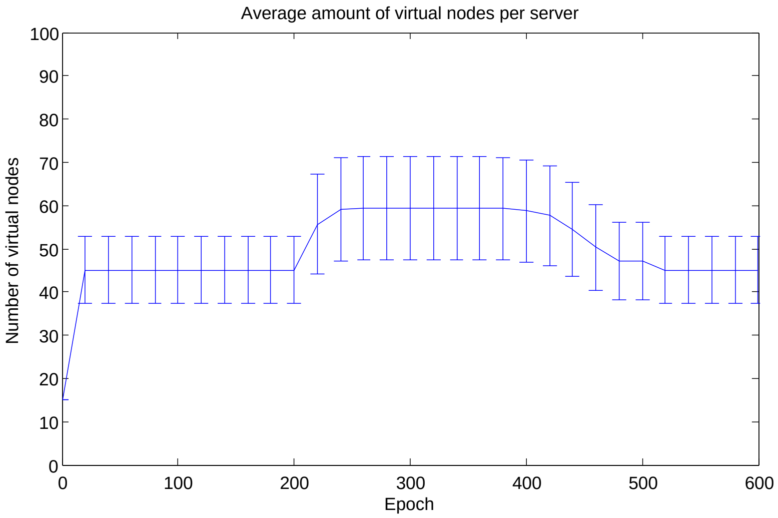


Figure 4.10: Large-scale scenario: total amount of virtual nodes in the system over time when the queries are evenly distributed among applications

for a minimum number of 2, 3, 4 replicas respectively. We first consider the case where the load is evenly distributed among the applications. Following the Pareto distribution properties, a small amount of virtual nodes are responsible for a large amount of queries. These virtual nodes become wealthier thanks to their high popularity, and they are able to replicate to one or several servers in order to handle the increasing load. Therefore, the total amount of virtual nodes is adjusted to the query load, as depicted in Figure 4.10. The number of virtual nodes remains almost constant during the high query load period. This is explained as follows: for robustness, replication is only initiated by a high query load. However, a replicated virtual node can survive even with a small number of requests before committing suicide. Therefore, the number of virtual nodes decreases when the query load is significantly reduced. Finally, around epoch 400, the balance of the additional replicated virtual nodes becomes negative and they commit suicide. More importantly, one application does not impact the performance of the others despite the variations in the total query load, as depicted in Figure 4.11. Moreover, the query load is balanced among the servers, as depicted in Figure 4.12, and the small variation in the virtual rent prices of the servers confirms that the equilibrium objective is reached, as depicted in Figure 4.13.

In the second case, where 4/7, 2/7 and 1/7 fractions of the total query load are attracted by application 1 (virtual ring 0), 2 (virtual ring 1) and 3 (virtual ring 2) respectively, the uneven query load does not impact the performance of the other applications as depicted in Figure 4.14, and the load is balanced among the servers as well, as depicted in Figure 4.15. However, the variation of the average load per server is greater than in the case where the load is evenly distributed. As most of the query load (i.e., 4/7 of the total query load) is attracted by the application with the smallest minimum number of replicas (i.e., application 1 on virtual ring 0 has a minimum number of 2

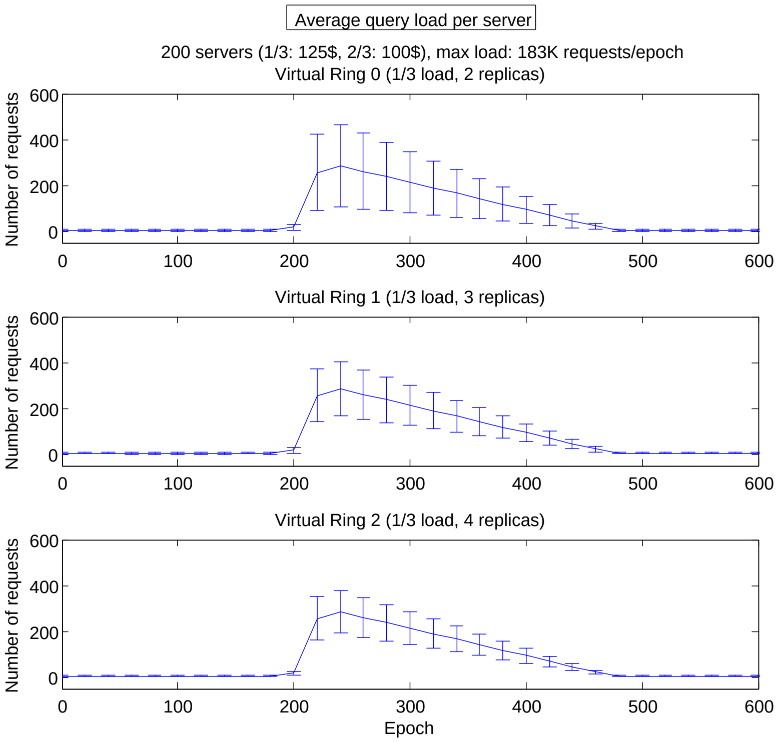


Figure 4.11: Large scale scenario: average query load per virtual ring per server over time when the queries are evenly distributed among applications

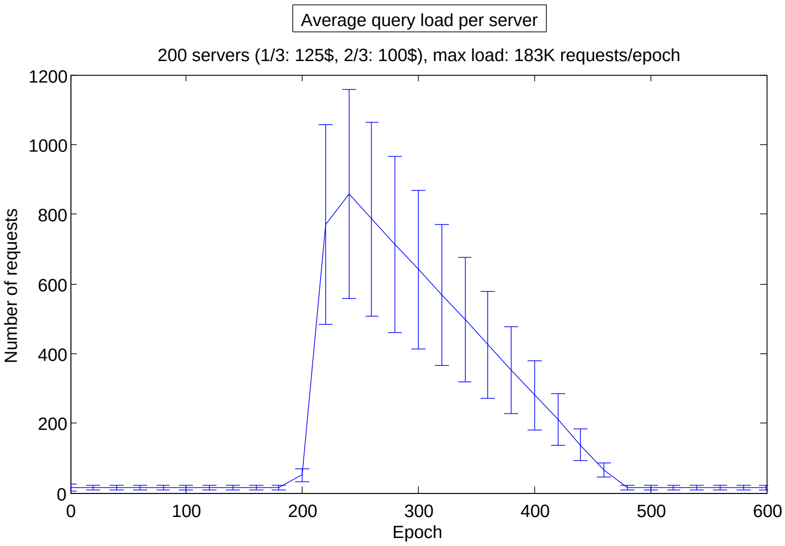


Figure 4.12: Large scale scenario: average query load per server over time when the queries are evenly distributed among applications

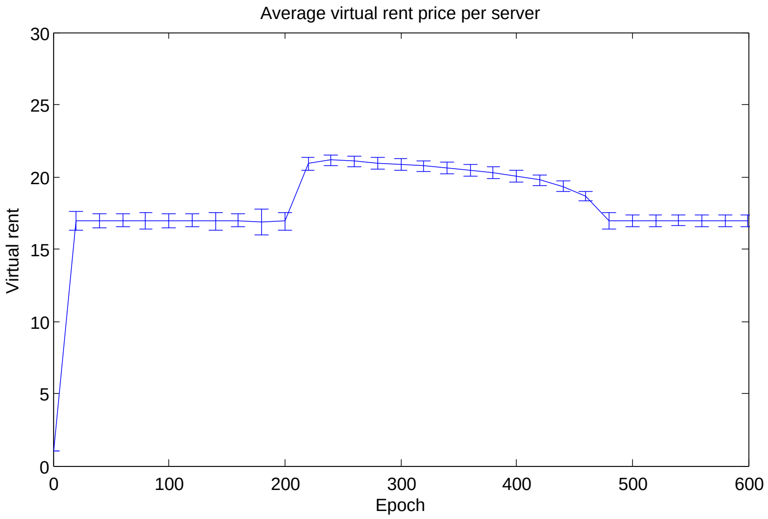


Figure 4.13: Large scale scenario: average virtual rent price per server over time when the queries are evenly distributed among applications

replicas), the replication process of the popular partitions must last longer to better balance the load among the servers.

4.7.5 Scalability of the Approach

Initially, we investigate the scalability of the approach regarding the storage capacity. For this purpose, we assume the arrival of insert queries that store new data into the cloud. The insert queries are again distributed according to Pareto(1, 50). We allow a maximum partition capacity of 256MB after which the data of the partition is split into two new ones, so that each virtual node is always responsible for up to 256MB of data. The insert query rate is fixed and equal to 2000 queries per epoch, while each query inserts 500KB of data. We employ the large-scale scenario parameters, but with the number of servers $N = 100$ and 2 racks per room in this case. The initial number of partitions is $M = 200$. We fill the cloud up to its total storage capacity. As depicted in Figure 4.16, our approach manages to balance the used storage efficiently and fast enough so that there are no data losses for used capacity up to 96% of the total storage. At that point, virtual nodes start not fitting to the available storage of the individual servers and thus they cannot migrate to accommodate their data.

Next, we consider that the query rate to the cloud is not distributed according to Poisson, but it increases with the rate of 200 queries per epoch until the total bandwidth capacity of the cloud is saturated. In this experiment, real rents of servers are uniformly distributed in [1$, 100$]. Now, our approach for selecting the destination server of a new replica is compared against two other rather basic approaches:

- **random**: a random server is selected for replication and migration, as

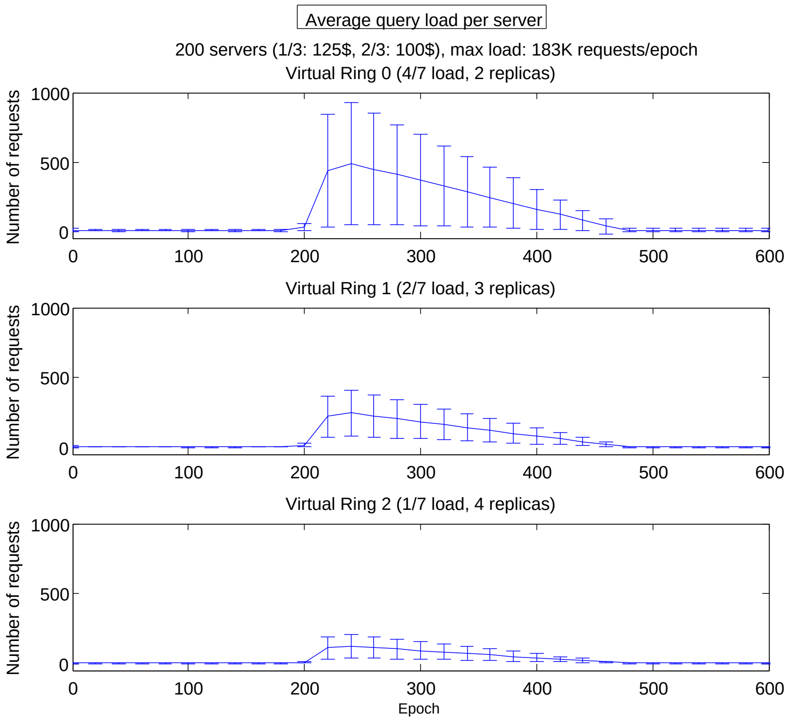


Figure 4.14: Large scale scenario: average query load per virtual ring per server over time when 4/7, 2/7, 1/7 of the queries are attracted by application 1, 2, 3 respectively

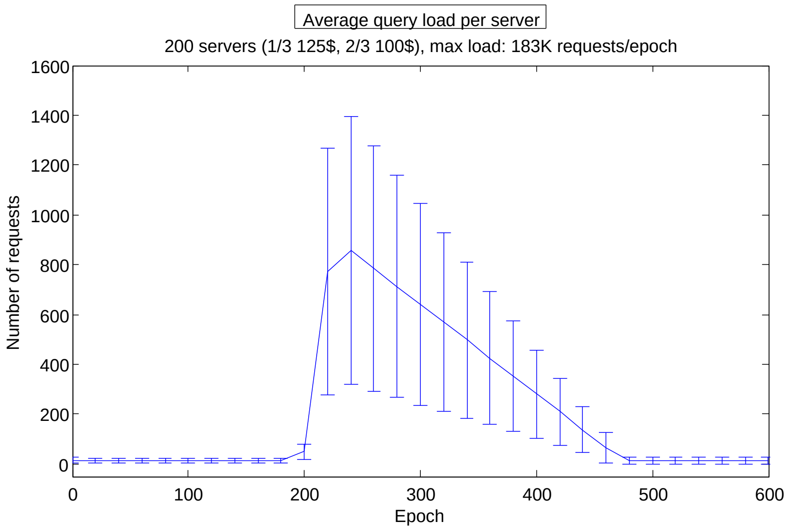


Figure 4.15: Large scale scenario: average query load per server over time when 4/7, 2/7, 1/7 of the queries are attracted by application 1, 2, 3 respectively

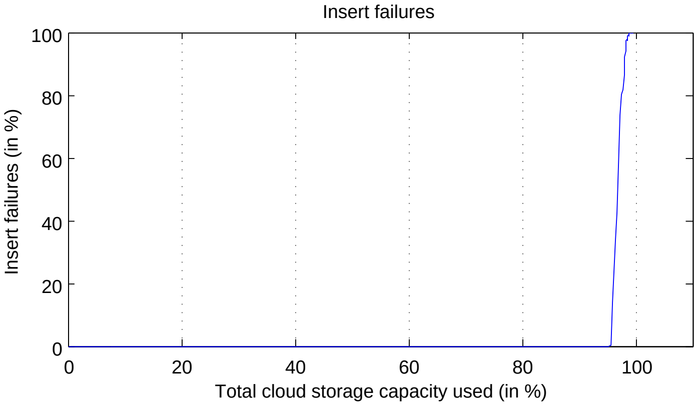


Figure 4.16: Storage saturation: insert failures

long as it has the available bandwidth capacity for migration and replication, and enough storage space.

- **greedy**: the cheapest server is selected for replication and migration, as long as it has the available bandwidth capacity for migration and replication, and enough storage space.

As depicted in Figure 4.17, our approach (referred to as "economic") outperforms the simple approaches regarding the amount of dropped queries having the bandwidth of the cloud completely saturated. Specifically, only 5% of the total queries are dropped at this worst case scenario. Therefore, our approach multiplexes the resources of the cloud very efficiently.

4.8 Implementation and Experimental Results in a Real Testbed

We have implemented a fully working prototype of *Skute* on top of Project Voldemort [40], which is an open source implementation of Dynamo [78] written in Java. Servers are not synchronized and no centralized component is required. The epoch is considered to be equal to 30 seconds. We have implemented a fully decentralized board based on a gossiping protocol, where each server exchanges its virtual rent price periodically with a small ($log(N)$, where N is the total number of servers) random subset of servers. Routing tables are maintained using a similar gossiping protocol for routing entries. The periods of these gossiping protocols are assumed to be 1 epoch. In case of migration, replication or suicide of a virtual node, the hosting server broadcasts the routing table update using a distribution tree leveraging the geographical topology of the servers.

Our testbed consists of $N = 40$ *Skute* servers, hosted by 8 machines (OS: Debian 5.0.3, Kernel: 2.6.26-2-amd64, CPU: 8 core Intel Xeon CPU E5430 @ 2.66GHz, RAM: 16GB) with Sun Java 64-Bit VMs (build 1.6.0_12-b04) and connected in a 100 Mbps LAN. According to our scenario, we assume a *Skute* data cloud spanning across 4 European countries with 2 datacenters per coun-

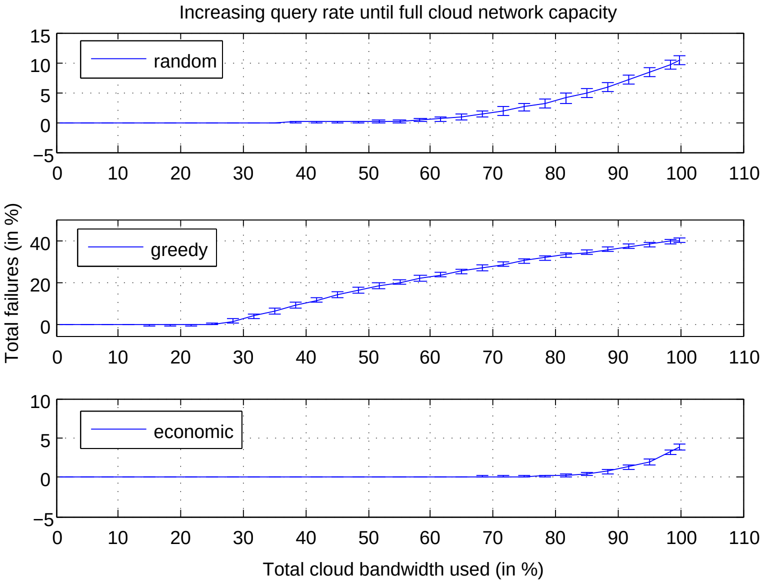


Figure 4.17: Network saturation: query failures

try. Each datacenter is hosted by a separate machine and contains 5 *Skute* servers, which are considered to be at the same rack. That is, the physical distance between servers in our testbed is not fully representative of a real deployment across 4 countries, where the latency between servers would be greater. We consider 3 applications, each of $M = 50$ partitions, with a minimum required availability of 2, 3 and 4 replicas respectively. 250000 data items of 10KB have been evenly inserted in the 3 applications. We generate 100 data requests per second using a Pareto(1,50) key distribution, denoted as application traffic. We refer as control traffic to the data volume transferred for migrations, replications and the maintenance of the boards as well as the routing tables.

4.8.1 Verification of Simulation Results

We first validate our simulation results of Section 4.7, by repeating the experiment of Figure 4.14. Specifically, at second 460, the total request rate is suddenly multiplied by 60 (from 6 to 360 requests/sec) and then slowly (5 requests/sec) decreases back to the initial rate. As depicted in Figure 4.18, the system adapts to the sudden load change similarly to the simulated one.

4.8.2 Scalable Performance

Also, we measure the effectiveness of the dynamic replication scheme in terms of response time and throughput. To this end, we increase the number of concurrent users that continuously send read requests for a particular data

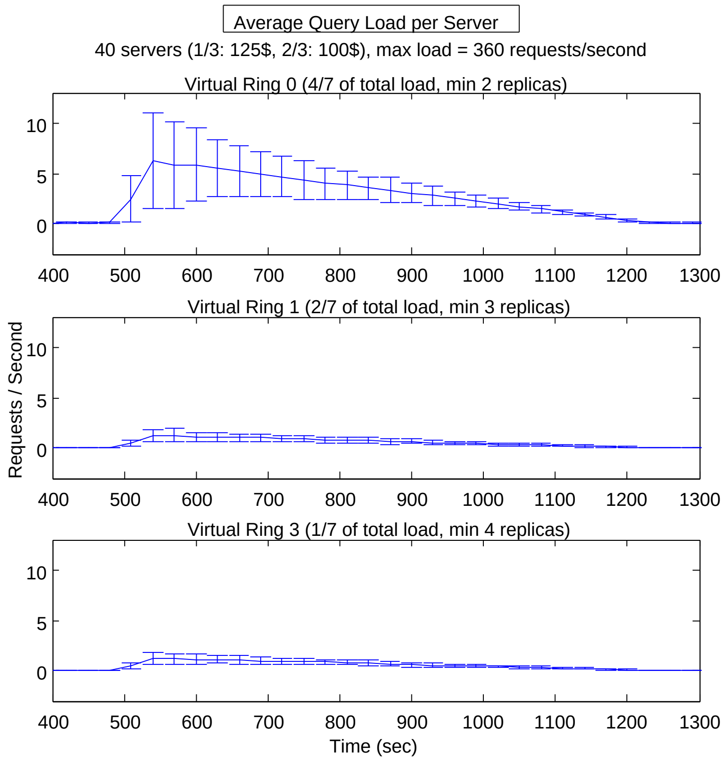


Figure 4.18: Average query load per virtual ring per server over time when 4/7, 2/7, 1/7 of the queries are attracted by application 1, 2, 3 respectively

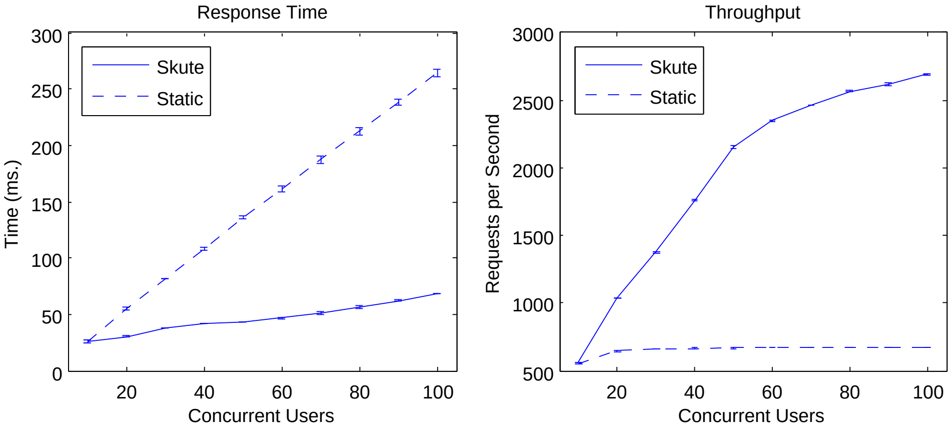


Figure 4.19: Left: Average response time (95-percentile). Right: Average throughput

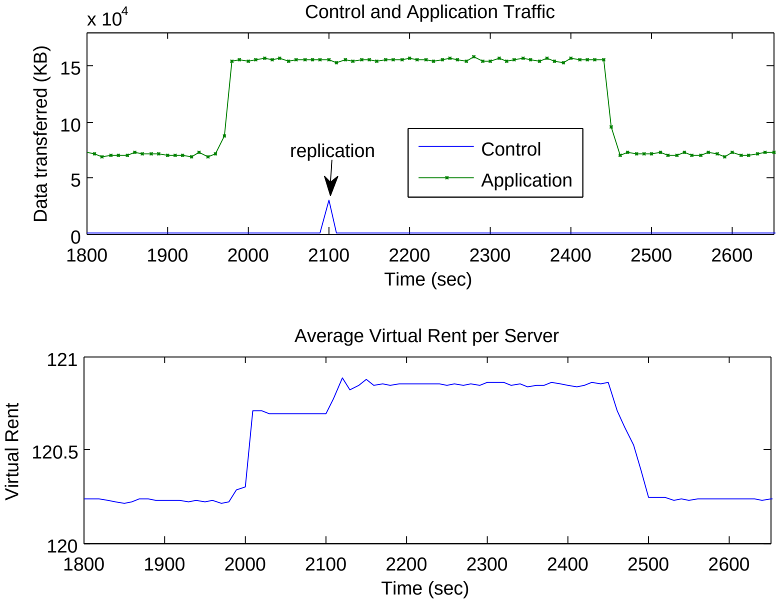


Figure 4.20: Top: Application and control traffic in case of a load peak. Bottom: Average virtual rent in case of a load peak.

item (i.e., the data popularity) from 0 to 100. As depicted in Figure 4.19, *Skute* achieves low response time and high throughput respectively, as compared to static replication with fixed number of replicas.

4.8.3 Adaptivity to Varying Load

We next evaluate the behavior of the system in case of a load peak. At second 1980, additional 100 requests per second are generated for a unique key. After 100 seconds, at second 2080, the popular virtual node hosting this unique key is replicated, as shown by the peak in the control traffic in Figure 4.20(top). Moreover, as depicted in Figure 4.20(bottom), the average virtual rent price increases during the load peak, as more physical resources are required to serve the increased number of requests. It further increases after the replication of the popular virtual node, because more storage is used at a server for hosting the new replica of the popular partition.

4.8.4 Adaptivity to Failure

Finally, the behavior of the system in case of a server crash is assessed. At second 2800, a *Skute* server collapses. As soon as the virtual nodes detect the failure (by means of the gossiping protocols), they start replicating the partitions hosted on the failed *Skute* server to satisfy again the minimum availability guarantees. Figure 4.21(top) shows that the replication process (as revealed by the increased control traffic) starts directly after the crash. Moreover, as depicted in Figure 4.21(bottom), the average virtual rent increases

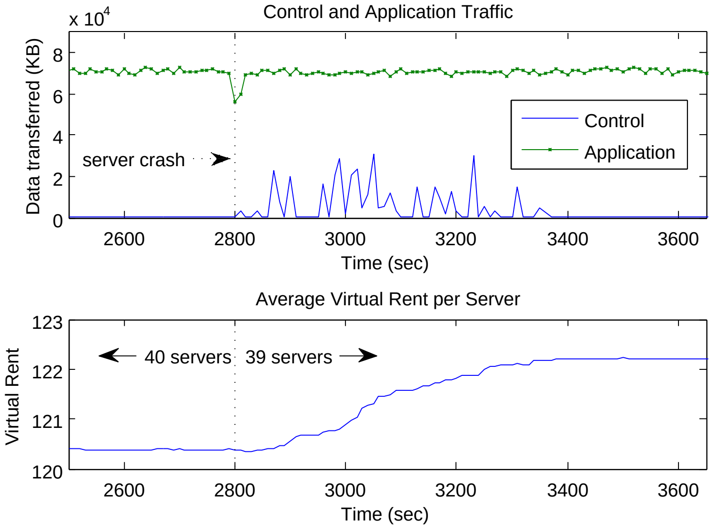


Figure 4.21: Top: Application and control traffic in case of a server crash. Bottom: Average virtual rent in case of a server crash.

during the replication process, because the same storage and processing requirements as before the crash, have to be now satisfied by fewer servers. Note that, in every case and especially when the system is at equilibrium, the control traffic is minimal as compared to the application one.

4.9 Potential Applications

So far, we have seen that *Skute* is able to provide high-availability guarantees by geographical diversity and query load-balancing by adaptive replica management. In this section, we explain why our approach could also provide an attractive alternative as a caching solution for large web applications. Typically, client requests first hit a reverse proxy, which is usually responsible for filtering the requests and serving directly static content. Dynamic content is handled by the application servers which have to query the database tier to complete the request. Thus, the main bottleneck of a large application often resides at the database tier. To this end, a caching tier is added to the architecture to decrease the load on the database. Then, mostly write operations contact the database. We propose that a memory-based *Skute* store is used as a caching layer. This solution has two main advantages as compared to common caching solutions, such as Memcached [26]: i) Adding or removing a new node in Memcached involves many cache entries invalidation, as opposed to *Skute*. Also, a single failure in Memcached may result in disabling entirely the caching tier with serious performance losses for cache restoration. ii) In Memcached, even if a popular data is cached, there is usually only one server responsible for serving the data. However, in *Skute*, adaptive data replication improves the read query performance.

Moreover, deploying the caching tier, across several datacenters may be ben-

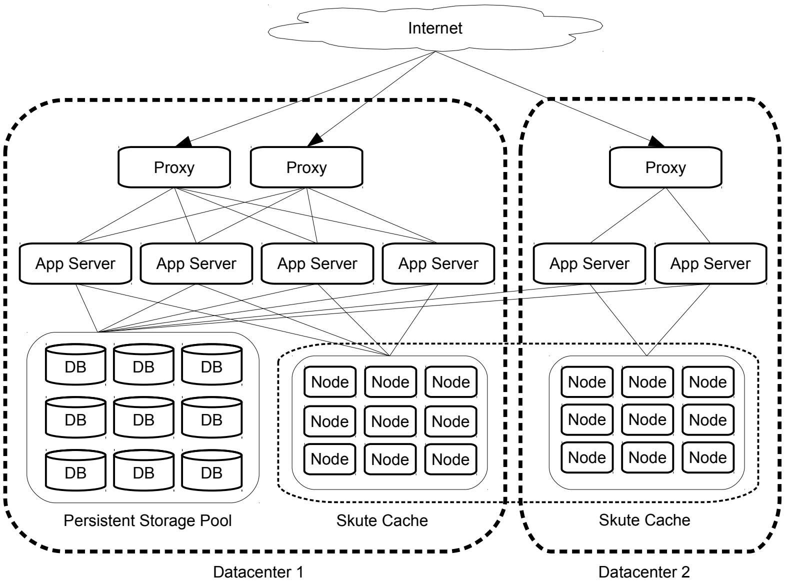


Figure 4.22: Architecture of a large web application using a *Skute* cache

eficial for large web applications for fault-tolerance and for moving the data closer to the clients. However, maintaining consistent storage at the database layer can be costly in terms of communication overhead. Our approach can serve as a geographically distributed in-memory caching tier, having the persistent storage residing at a single datacenter, as depicted in Figure 4.22. Read operations are mostly sent to the caching tier, while write ones are sent to the database. Therefore, application data can reside closer to the final clients and high data availability can be guaranteed.

Anycast Service In the case of a global service relying on anycast to serve clients closely to their geographical location, the content of the service should be replicated or mirrored to every edge server. For DNS services, a common practice is to use rsync over ssh to mirror the zones among the authoritative servers or to rely on the master-slave architecture provided directly by the DNS server. In order to maintain consistency, write operations are performed only at a single facility and mirrored to the others facilities. Obviously, when the master facility is unreachable, no write operation is possible. This might be a real problem for systems like CDNs where new DNS entries are continuously added or updated. Using *Skute* as the DNS server backend permits write operations to any facilities, while ensuring that every edge server has eventually a local replica of the zones.

4.10 Related Work

Dealing with network failure, strong consistency (which databases care of) and high data availability cannot be achieved at the same time [63]. High data

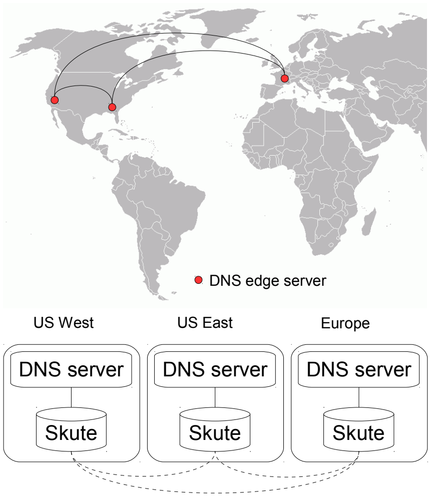


Figure 4.23: Architecture of an anycast DNS service

availability by means of replication has been investigated in various contexts, such as P2P systems [137, 113], data clouds, distributed databases [128, 78] and distributed file systems [95, 142]. In the P2P storage systems PAST [137] and Oceanstore [113], the geographical diversity of the replicas is based on random hashing of data keys. Oceanstore deals with consistency by serializing updates on replicas and then applying them atomically. In the distributed databases and systems context, Coda [142], Bayou [128] and Ficus [95] allow disconnected operations and are resilient to issues, such as network partitions and outages. Conflicts among replicas are dealt with different approaches that guarantee event causality. In distributed data clouds, Amazon Dynamo [78] replicates each data item at a fixed number of physically distinct nodes. Dynamo deals with load balancing by assuming the uniform distribution of popular data items among nodes through partitioning. However, load balancing based on dynamic changes of query load are not considered. In all the aforementioned systems, replication is employed in a static way, i.e., the number of replicas and their location are predetermined. Also, no replication cost considerations are taken into account and no geographical diversity of replicas is employed.

In [152], data replicas are organized in multiple rings to achieve query load-balancing. However, only one ring is materialized (i.e., has a routing table) and the other rings are accessible by iteratively applying a static hash function. This static approach for mapping replicas to servers does not allow to perform advanced optimizations, such as moving data close to the end user and ensuring the geographical diversity between replicas, or offering a different availability level per application/data item.

Some economic-aware approaches are dealing with the optimal locations of replicas. Mariposa [150] aims at latency minimization in executing complex queries over relational distributed databases, i.e., not primary-key access queries on which we focus. Sites in Mariposa exchange data items (i.e., migrate or replicate them) based on their expected query rate and their processing cost using combinatorial auctions, where winner determination is tricky and synchronization is required. In our approach, asynchronous individual decisions are taken by data items regarding replication, migration or deletion, so that high availability is preserved and dynamic load balancing is performed. Also, in [147], a cost model is defined for the factors (e.g., query load, network and storage capacity) that affect data and application migration for minimizing latency in replying queries.

On the other hand, in the Mungi operating system [100], a commodity market of storage space has been proposed. Specifically, storage space is lent by storage servers to users and the rental prices increase as the available storage runs low (similarly to congestion pricing in networks), forcing users to release unneeded storage. This approach does not take into account the different query rates for the various data items and it does not have any availability objectives.

In [105], an approach is proposed for reorganizing replicas evenly in case that new storage is added into the cloud, while minimizing data movement. Relocated data and new replicas are randomly assigned with higher probability to newer servers. However, this replication approach does not consider geographical distribution of replicas, differentiated availability levels to multiple applications, or data popularity.

In [59] and [77] efficient data management for the consistency of replicated data in distributed databases is addressed by an approach guaranteeing one-copy serializability in the former and snapshot isolation in lazy replicated databases (i.e., where replicas are synchronized by separate transactions) in the latter. In our case, we do not expect high update rates in a key-value store and therefore concurrent copy of changes to all replicas can be an acceptable approach.

4.11 Conclusion

In this chapter, we described *Skute*, a robust, scalable and highly-available key-value store that dynamically adapts to varying query load or disasters by determining the most cost-efficient locations of data replicas with respect to their popularity and their client locations. We experimentally proved that our approach converges fast to equilibrium, where as predicted by a game-theoretical model no migrations happen for steady system conditions. Our approach achieves net benefit maximization for application providers and therefore it is highly applicable to real business cases. We have built a fully working prototype in a distributed setting that clearly demonstrates the feasibility, the effectiveness and the low communication overhead of our approach. As a future work, we plan to investigate the employment of our approach for more complex data models, such as the one in Bigtable [71].

Chapter 5

Building Highly-Available and Scalable Cloud Applications

The service-oriented architecture (SOA) paradigm for orchestrating large-scale distributed applications offers significant cost savings by reusing existing services. However, the high irregularity of client requests and the distributed nature of the approach may deteriorate service response time and availability. Static replication of components in datacenters for accommodating load spikes requires proper resource planning and underutilizes the cloud infrastructure. Moreover, no service availability guarantees are offered in case of datacenter failures. In this chapter, we propose a cost-efficient approach for dynamic and geographically-diverse replication of components in a cloud computing infrastructure that effectively adapts to load variations and offers service availability guarantees. In our virtual economy, components rent server resources and replicate, migrate or delete themselves according to self-optimizing strategies. We experimentally prove that such an approach outperforms in response time even full replication of the components in all servers, while offering service availability guarantees under failures.

Significant achievements have been made for automated allocation of cloud resources. However, the performance of applications may be poor in peak load periods, unless their cloud resources are dynamically adjusted. Moreover, although cloud resources dedicated to different applications are virtually isolated, performance fluctuations do occur because of resource sharing, and software or hardware failures (e.g., unstable virtual machines, power outages, etc.). In this chapter, we propose a decentralized economic approach for dynamically adapting the cloud resources of various applications, so as to statistically meet their SLA performance and availability goals in the presence of varying loads or failures. According to our approach, the dynamic *economic fitness* of a Web service determines whether it is replicated or migrated to another server, or deleted. The economic fitness of a Web service depends on its individual performance constraints, its load, and the utilization of the resources where it resides. Cascading performance objectives are dynamically calculated for individual tasks in the application workflow according to the user requirements.

By fully implementing our framework, we experimentally prove that our adaptive approach statistically meets the performance objectives under peak load periods or failures, as opposed to static resource settings.

5.1 Introduction

Cloud computing is deemed to replace high capital expenses for infrastructure with lower operational ones for renting cloud resources on demand by the application providers. However, with static resource allocation, a cluster system would be likely to leave 50% of the hardware resources (i.e., CPU, memory, disk) idle, thus baring unnecessary operational expenses without any profit (i.e., negative value flows). Moreover, as clouds scale up, hardware failures of any type are unavoidable.

A successful online application should be able to handle traffic spikes and flash crowds efficiently. Moreover, the service provided by the application needs to be resilient to all kinds of failures (e.g., software stales, hardware, rack or even datacenter failures, etc.). A naive solution against load variations would be static over-provisioning of resources, which would result into resource underutilization for most of the time. Resource redundancy should be employed to increase service reliability and availability, yet in a cost-effective way. Most importantly, as the size of the cloud increases its administrative overhead becomes unmanageable. The cloud resources for an application should be self-managed and adaptive to load variations or failures.

With the emergence of the cloud computing paradigm, avoiding high capital investment for infrastructure becomes viable. Lower operational expenses are expected for renting cloud resources on demand by the application providers. Although achievements in automated cloud resource provisioning were enough for the first wave of "best effort" application deployments, adaptive resource allocation for satisfying the performance and availability objectives of mission-critical application remains an open issue. With static resource allocation (based on resource planning and over-provisioning), a cluster system would be likely to leave 50% of the hardware resources (i.e., CPU, memory, disk) idle, thus baring unnecessary operational expenses without any profit (i.e., negative value flows). Moreover, as described in [79], the performance of multiple identical virtual machines may greatly vary, and thus might drastically reduce the performance of a distributed application. On the other hand, as clouds scale up, software and hardware failures of any type (e.g., software stales, virtual machines go "wonky", i.e become unstable), hardware or rack or even datacenter failures, etc.) are unavoidable and often *spatially correlated* [130]. Resource redundancy should be employed to increase service reliability and availability, yet in a cost-effective way. Another concern is that, as the size of the cloud increases, its administrative overhead becomes unmanageable.

In this chapter, we focus on cost-effective autonomic resource allocation, so as to adaptively satisfy service level agreements (SLAs) for performance and availability statistical guarantees against load variations and software / hardware failures. We propose a middleware ("Scattered Autonomic Resources", referred to as *Scarce*) that performs supple sharing to avoid stranded and underutilized computational resources and dynamically adapts to changing conditions, such as failures, load variations or "wonky" servers or virtual machines. As our framework works indifferently on top of virtualized and/or physical servers, henceforth, we use the terms *server* and *virtual machine* interchangeably, unless stated otherwise. Our middleware simplifies the development of online applications composed of multiple independent components (e.g., web services) following the Service Oriented Architecture (SOA) principles. We consider a *virtual economy*, where components are treated as individually rational entities that rent computational resources from servers, and migrate, replicate or exit according to their economic *fitness*. This fitness expresses the difference between the utility offered by a specific application component and the cost for retaining it in the cloud. The server rent price is an increasing function of the utilization of server resources. Moreover, components of a certain application are dynamically replicated to *geographically-diverse* servers according to the availability requirements of the application.

The economic approach and the geographical diversity of components along with their ability to migrate or replicate are directly inspired by *Skute* as described in Chapter 4, such that a component in *Scarce* is treated similarly to a virtual node in *Skute*. However, in this chapter, we deal with applications where components have dependencies and performance constraints among them, contrary to virtual nodes. Moreover, *Scarce* focuses on practical issues occurring when deploying an application on a cloud infrastructure: placing the components so as to offer robustness and performance guarantees on top of an elastic infrastructure with unpredictable performance becomes challenging. Our approach combines the following unique characteristics:

- Adaptive adjustment of cloud resource allocation in order to statistically satisfy response time or availability SLA requirements.
- Cost-effective resource allocation and component placement for minimizing the operational costs of the cloud application.
- Detection and removal or replacement of stale cloud resources.
- Component replication and migration for accommodating load variations and for supple load balancing.
- Decentralized self-management of the cloud resources for the application.
- Geographically-diverse placement of clone component instances.

Having implemented a full prototype of our approach, we experimentally prove that it effectively accommodates load spikes, it satisfies compliance to the SLA response-time requirements, it cost-effectively utilizes the cloud resources, and it provides a dynamic geographical replica placement without thrashing. We finally reveal the *trade-off* between cost-effectiveness and meeting strict SLA requirements; the latter may necessitate a more conservative (over-provisioning) resource allocation approach.

The remainder of this chapter is organized as follows: in Section 5.2, we present a motivating example application. In Section 5.3, we describe our economic approach for autonomic component replica management. Section 5.5 describes how SLAs are propagated. In Section 5.6, we describe how the components dynamically adapt their resources to honor their SLA. Subsection 5.6.1 describes how “wonky”and cloud resources are detected and removed. In Section 5.7, we present our experimental results. In Section 5.8, we overview the related work and, finally in Section 5.9, we conclude our work.

5.2 Motivation

Elastic platforms are becoming more and more popular and start to be a viable alternative to host distributed applications and web applications in particular. In a typical cloud infrastructure, a user can rent virtual machines (VM) and allocate dedicated resources (such as CPU cores, RAM, disk space, etc.) to them in order to closely match the application needs. Moreover, through the cloud infrastructure application programming interface (API), the user is able to programmatically increase or decrease the resources allocated to a virtual machine, and can also start new VMs or stop unused ones. The virtualization technologies that have greatly contributed to the success of cloud computing also come with some drawbacks. In effect, a user does not have full control over the underlying infrastructure. For example, in today’s public cloud infrastructures, such as Amazon EC2 [1] or Rackspace Cloud Servers [41], a user does not have the possibility to choose on which physical server a VM will be started. Moreover, a VM may migrate during its lifetime from one physical server to another. Also, a user has no control over the virtual machines that are collocated on the same physical server. This may have a large performance impact, if for example, a collocated VM performs an I/O intensive work. Figure 5.1 shows an architectural view of a distributed application hosted by a cloud computing infrastructure. The application consists of 5 replicated components (*Comp 1* to *Comp 5*) and spans 4 VMs and 3 physical servers. In such elastic infrastructures, each physical server hosts several VMs. As the application owner has usually no control on where its VMs are hosted, the application responsiveness may suffer from an overloaded VM of another customer hosted at the same physical server as the one employed by the application VM.

A well-engineered cloud application should be able to detect slow or “wonky” VMs and react accordingly. A desirable reaction would be to first gradually redirect the traffic for the components of the “wonky” VM to their replica components elsewhere. Second, if the overall performance of the application has suffered considerably, a new VM should be started in order to take over the redirected traffic. At some point, the “wonky” VM will not receive any traffic and can safely be removed, thus saving rental costs.

5.2.1 Running Example

Building an application that both provides robust guarantees against failures (hardware, network, etc.) and handles dynamically load spikes is a non-

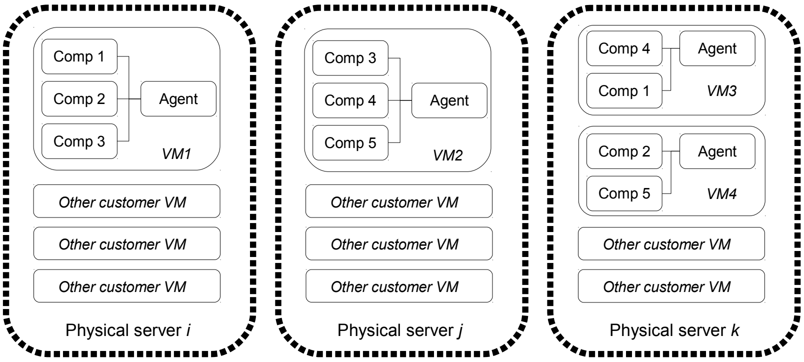


Figure 5.1: Overview of a distributed application composed of 5 components (i.e., *Comp 1* to *Comp 5*) deployed on a cloud computing infrastructure

trivial task. As a running example, we have developed a simple web application for selling e-tickets (print@home) composed of 4 independent components:

- A web front-end, which is the entry point of the application and serves the HTML pages to the end user.
- A user manager for managing the profiles of the customers. The profiles are stored in an highly scalable, eventually consistent, distributed, structured key-value store [4].
- A ticket manager for managing the number of available tickets for an event. This component uses a relational database management system (MySQL).
- An e-ticket generator that produces e-tickets in PDF format (print@home).

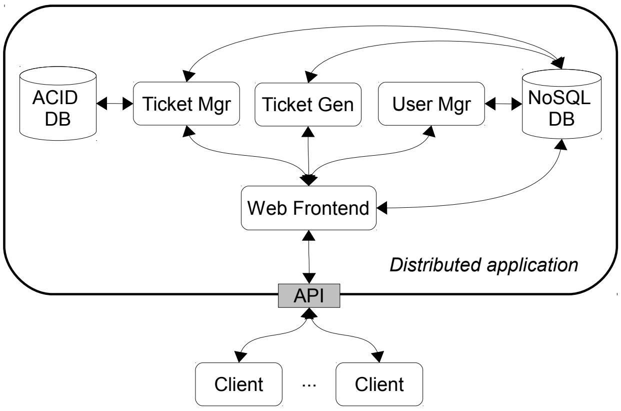


Figure 5.2: A distributed application using different components.

Each component can be regarded as a stateless, standalone and self-contained web service. Figure 5.2 depicts the application architecture. A token (or a session ID) is assigned to each customer's browser by the web front-end and is passed to each component along with the requests. This token is used as a key in the key-value database to store the details of the client's shopping cart, such as the number of tickets ordered. Note that even if the application uses the concept of sessions, the components themselves are stateless (i.e., they do not need to keep an internal state between two requests).

This application is highly sensitive to traffic spikes, when, for example, tickets for a concert of a famous band are sold. If the spike is foreseeable, one wants to be able to add spare servers that will be used transparently by the application for a short period of time, without having to reconfigure the application. After this period, the servers have to be removed transparently to the end users. As this application is business-critical, it needs to be deployed on different geographical regions, hence on different datacenters.

5.3 Scarce: the Quest of Autonomic Applications

5.3.1 The Approach

We consider applications formed by many independent and stateless components that interact among each other to provide a service to the end user, as in the Service Oriented Architecture (SOA) paradigm. A component is self-managing, self-healing and is hosted by a server (or a virtual machine), which in turn is allowed to host many different components. A component can stop, migrate or replicate to a new server according to its load or availability. The approach to maintain high availability is explained in Section 5.4.

5.3.2 Server Agent

The server agent is a special component that resides at each server and is responsible for managing the resources of the server according to our economic-based approach, as shown in Figure 5.3. Specifically, this agent is responsible for starting and stopping the components of the various applications at the local server, as well as checking the "health" of the services, e.g., by verifying if the service process is still running, or by issuing a test request and checking that the corresponding reply is correct. The agent knows the properties of every service that composes the application, such as the path of the service executable, and its minimum and maximum replication factor. This knowledge is acquired when the agent starts, by contacting another agent, referred to as "bootstrap agent". Any running agent participating in the application cluster can act as a bootstrap agent.

During the startup phase, the agent also retrieves the current routing table from the bootstrap agent. The routing table provides a mapping between services and servers (*cf.* Section 5.3.3). The number of replicas of a service and their placement are handled by a distributed optimization algorithm autonomously executed by the agents.

In an untrustworthy environment, where a server agent may be malicious, the functionality of decision making could be implemented directly in the

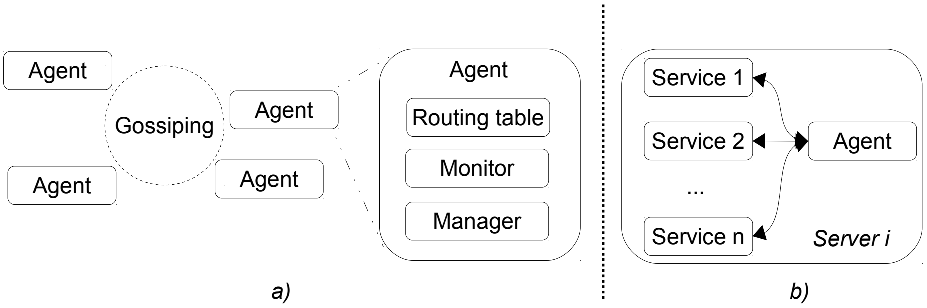


Figure 5.3: *a)* Agents communicate using a gossiping protocol *b)* A server or a virtual machine hosts many services along with an agent responsible for managing them

component itself. While being robust to strategic behaviors of server agents, this approach tends to waste resources, as every component would have to perform the tasks of a server agent (i.e., maintaining the routing table, gossiping, etc.).

We assume, as in Chapter 4, that a server belongs to a rack, a room, a datacenter, a city, a country and a continent. Note that finer or coarser geographical granularity could also be considered, especially in a cloud environment where the rack and room information may not be available. A label of the form "continent-country-city-datacenter-room-rack-server" is attached to each server in order to precisely identify its geographical location. For example, a possible label for a server located in a data center in London could be "EU-UK-LON-D1-C03-R11-S07".

5.3.3 Routing Table

Instead of using a centralized repository for locating services, such as a UDDI registry (uddi.xml.org), each server keeps locally mappings (i.e., a routing table) between components and servers. This routing table (e.g., Table 5.1) is maintained by a gossiping algorithm (see Figure 5.3), where each agent contacts a random subset [$log(N)$ where N is the total number of servers] of remote agents and exchanges information about the services running at their respective server.

Table 5.1: The local routing table

component	*servers*
component 1	server A, server B
component 2	server B, server C
component 3	server A

A component may be hosted by several servers, therefore we consider 4 different policies that a server s may use for choosing the replica of a component:

- a **proximity-based** policy: thanks to the labels attached to each server, the geographically nearest replica is chosen;

- a **rent-based** policy: the least loaded server is chosen; this decision is based on the rent price of the servers.
- a **random-based** policy: a random replica is chosen.
- a **net benefit-based** policy: the geographically closest and least loaded replica. For every replica of the component residing at server *j*, we compute a weight:

$$wp_j = \frac{\varepsilon + proximity(s,j) - rent_j}{\sum_{i \in replicas}(\varepsilon + proximity(s,i) - rent_i)}, \tag{5.1}$$

  where $0 < \varepsilon << 1$ is a very small positive, $proximity(s,j)$ returns a value corresponding to the geographical proximity of s and j according to their labels (the greater the value, the closer the servers) and $rent_j$ is the virtual rent of the server j, which gives an approximation of its load, as will be described in Section 5.3.4. This weight represents the probability that the replica j will be chosen.

5.3.4 Economic Model

Service replication should be highly adaptive to the processing load and to failures of any kind in order to maintain high service availability. To this end, each component is treated by the server agent as an individual optimizer that acts autonomously so as to ascertain the availability guarantees pre-specified by the SLA and to *balance* its economic fitness. Time is assumed to be split into epochs. At every epoch, the server agent verifies from the local routing table that the minimum requirement on the number of replicas for every component is satisfied; thus, no global or remote knowledge is required. If the required availability level is not satisfied and if the service is not already running locally, the agent starts the service. When the service has started, the server agent informs all others by using a hierarchical broadcast to update their respective routing tables.

At each epoch, a service pays a *virtual rent* to the servers where it is running. The virtual rent corresponds to the usage of the server resources, such as CPU, memory, network, disk I/O and space. A service may be replicated or migrated to another server, or stopped by the server agent. These decisions are made based on the service demand, the renting cost and the maintenance of high availability upon failures. There is no global coordination and each server agent behaves independently for each hosted service. Only one replica of a service is allowed to be stopped at the same epoch by employing the Paxos [114] distributed consensus algorithm. The virtual rent of a server is updated at the beginning of a new epoch by the server agent. The price of the other servers participating in the application cluster are updated by the same gossiping algorithm that is used to maintain the routing table.

The actions (i.e., replication, migration, stop) performed by the server agent on behalf of a component c hosted at a server s are directly related to the economic fitness or *balance* of the component c, which is given by:

$$balance_c = utility_c - rent_s\,, \tag{5.2}$$

The utility of a component corresponds to the value that it creates for the various applications that employ it and it can be safely assumed to be an increasing function of the server resources utilized by the component. We denote as x_c the usage percentage of the server resources by the component c and as x_s^c the *contention level* of the server s for a given application component c. x_c can be calculated as follows:

$$x_c = \frac{w_c \cdot cpu_c + w_m \cdot mem_c + w_n \cdot net_c + w_d \cdot disk_c}{w_c + w_m + w_n + w_d}, \tag{5.3}$$

where cpu_c, mem_c, net_c, $disk_c$ are the component usage percentages for CPU, memory, network and disk respectively. x_c is normalized to $[0,\ 1]$. w_c, w_m, w_n, $w_d \in [0,\ 1]$ are weights to adapt to different kinds of application components (CPU-intensive, I/O-intensive, etc.): a high CPU, memory, network or disk usage corresponds to a high value for w_c, w_m, w_n, w_d respectively. The resource usage of a component can be determined using standard process performance tools available on Linux, such as top [49], iotop [21], lsof [25], sysstat [48], NetHogs [34] or by reading the appropriate files under `/proc`.

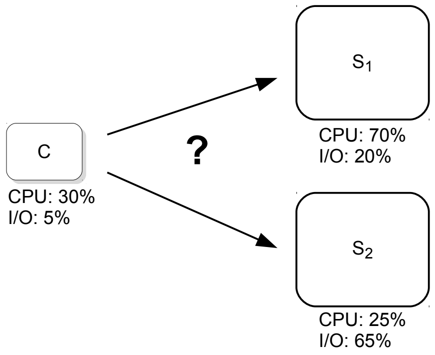


Figure 5.4: Contention level of servers with respect to a component

x_s^c is calculated similarly to x_c by employing the resource utilization percentages for the total server instead of the single component ones:

$$x_s^c = \frac{w_c \cdot cpu_s + w_m \cdot mem_s + w_n \cdot net_s + w_d \cdot disk_s}{w_c + w_m + w_n + w_d}, \tag{5.4}$$

where cpu_s, mem_s, net_s, $disk_s$ are the server usage percentages for CPU, memory, network and disk respectively, and w_c, w_m, w_n, w_d are the weights of the component c. That is, x_s^c, which is normalized to $[0,\ 1]$, depends not only on the server usage but also on the kind of the component (CPU-intensive, I/O-intensive, etc.). For example, imagine a component c that has to migrate to another server and can choose between two servers, s_1 and s_2. Let us consider a simple scenario, where only CPU and disk I/O usage are taken into account, as depicted in Figure 5.4. Both servers having the same total resource usage of 45%, the component should give preference to the server that will offer the best performance (i.e., the least contention) for its kind of resource utilization, namely s_2 because $x_{s_2}^c < x_{s_1}^c$. To this end, the utility of a component c residing at server s is assumed to be given by the following convex formula:

$$utility_c = \frac{x_c - x_s^*}{(C - x_c)^2}, \tag{5.5}$$

where $C > 1$ is a constant determining the starting point of the fast-increasing part of the curve. $utility_c$ is normalized to $[0,\ k]$ with $k > 1$ in order the utility of c to be greater than the rent price of the server s, such that the component could be wealthy enough to afford the rent of another server in case of replication. The selection of the parameters k and C determines the reactivity of the balance of a component with respect to its resource usage. Typical values are $k = C = 5$. x_s^* is a component utilization threshold that determines when the component residing at s is economically *fit-enough* to replicate, i.e.,:

$$x_s^* = \frac{srvMinUsage}{|components_s|}, \tag{5.6}$$

where $|components_s|$ is the number of components running at the server and $srvMinUsage$ is a percentage threshold denoting a *soft limit* for server utilization, e.g., 25%. The utility function is chosen so that it grows exponentially to the usage and it is 0 for $x_c = x_s^*$. The virtual rent paid by the component c to the server s is given by:

$$rent_s = conf_s \cdot x_s\,, \tag{5.7}$$

where $conf_s \in [0,\ 1]$ is a subjective estimation of the server quality and reliability based on technical factors (hardware quality, datacenter connectivity, redundancy, etc.) as well as non-technicals ones (e.g., political and economical stability of the country hosting the server, etc.).

Based on the $balance_c$, at the beginning of a new epoch, a component may:

- **migrate or stop:** if it has negative balance for the last f epochs. First, the component calculates its availability without itself. If the availability is satisfactory, the component stops. Otherwise, it tries to find a less expensive (i.e., busy) server that is closer to the client locations (according to maximization formula (5.9)). To avoid oscillations of a replica among servers, the migration is only allowed if the following *migration conditions* apply:
  - The minimum availability is still satisfied using the new server,
  - the absolute price difference between the current and the new server is greater than a threshold,
  - the x_s of the current server s is above the soft limit $srvMinUsage$.
- **replicate:** if it has positive balance for the last f epochs, it may replicate. For replication, a component has also to verify that it can afford the replication by having a positive balance b' for consecutive f epochs:

  $$b' = balance_c - (1 + \varphi) \cdot rent_{s'}$$

  where $rent_{s'}$ is the current virtual rent of the candidate server s' for replication (randomly selected among the top-k ones ranked according to the formula (5.9)), while the factor $1 + \varphi$ accounts for a $\varphi{\cdot}100\%$

increase at this rent price in the next epoch due to the potentially increased usage of the candidate server (an upper bound of $\varphi = 0.2$ can typically be assumed). This action aims to distribute to load of the current server towards another one located closer to the clients. Thus, it tends to decrease the processing and network latency of the requests for the component.

5.4 Maintaining High-Availability

Server or component failures or network partitioning may unexpectedly occur at any time and they are often spatially-correlated. As estimating the probability of each server to fail necessitates access to a large set of historical data and private information of the server, we adopt the approach of 4.4.3 for maintaining high availability by geographically-diverse placement of component replicas. The availability of a service *i* is defined as the sum of *diversities* between each distinct pair of servers, i.e.,:

$$avail_i = \sum_{i=0}^{|S_i|} \sum_{j=i+1}^{|S_i|} conf_i \cdot conf_j \cdot diversity(s_i, s_j) \tag{5.8}$$

where $S_i = (s_1, s_2, \ldots, s_n)$ is the set of servers hosting replicas of the service i and $conf_i$, $conf_j \in [0,\ 1]$ are the confidence levels of servers i, j. The diversity function returns a value calculated based on the geographical distance among each server pair. This distance can be represented as a n- bit number, having each bit corresponding to the n location parts of a server, e.g., continent, country, city, data center, room rack, server etc. The most significant bit (leftmost) represents the wider enclosing geographical location (e.g., the continent), while the least significant bit (rightmost) represents the server. When two servers are not in the same location part, their corresponding *diversity* bit is set to 1, otherwise to 0. Once a bit has been set to 0, all less significant bits are also set to 0. For example, two servers belonging to the same data center but located in different rooms cannot be in the same rack, thereby all bits after the third bit (data center) have to be 0. The proximity number would then look like this:

cont	coun	city	datac	room	rack	serv
1	1	1	0	0	0	0

A binary "NOT" operation is then applied to the proximity to get the diversity value:

$$\overline{1110000} = 0001111 = 15(decimal)$$

The diversity values of the server pairs are summed up, because having more replicas in distinct servers always results in increased availability regardless of their location. A component knows the locations of its replicas by the local routing table at the server where it is hosted.

The availability of a component should always be kept above a minimum level *th*, which is derived by the SLA. When the availability of a component falls below *th*, a new service instance should be started (i.e., replicated) at a new server. The best candidate server is selected so as to maximize the *net benefit* between the diversity of the resulting set of replica locations for the service and the virtual rent of the new server, i.e.,

$$\sum_{k=1}^{|S_i|} g_j \cdot conf_j \cdot diversity(s_k, s_j) - rent_j\,, \tag{5.9}$$

where $rent_j$ is the virtual rent price of candidate server j. g_j is a weight related to the proximity (i.e., inverse average diversity) of the server location to the geographical distribution of the client requests for the service (*cf.* 4.4.3). Note that client requests may come from other components. As a result, the components will tend to replicate closer to the components that heavily rely on the services of the former. The components rank servers according to their net benefit (5.9) and randomly choose the target for replication among the top-k ones for avoiding server congestion. Note that the same approach according to (5.9) is used for choosing the candidate server for component migration.

5.5 Meeting SLA Performance Guarantees

Autonomic migration or replication of the application components in order to utilize in a fair manner the available resources may not always be good enough to guarantee acceptable end-to-end service quality. If the latency of client requests is not satisfactory and the allocated resources to the application are not underutilized, one solution is to give more resources to the application, e.g., by increasing the number of cores of a virtual machine (VM), or by starting a new VM.

To this end, our framework is able to manage the physical resources dedicated to the application based on a SLA defined by the application owner. If the SLA is not met, then the framework asks for more resources via the cloud API. Or, if the application easily honors the SLA, it can remove some extra resources.

5.5.1 Cascading Performance Constraints

The application owner requires the compliance of the performance to certain constraints pre-specified in a SLA, e.g., an upper bound on the response time for a service request. In case of complex applications that consist of many components that have dependencies on each other (as the one depicted in Figure 5.5), it is not always possible to comply to the SLA-driven performance constraints, unless the latter are individually set to each component constituting the application. However, the performance constraints can be directly derived by the SLA only for the entry component (i.e., the one that receives the user requests) of the application; derivation of the performance constraints for the other components by the SLA would necessitate *a priori* knowledge of the application internals, e.g., the exact execution workflow, the hardware resources allocated to each component, the component computational needs, etc. Moreover, the performance constraints for each component should change over time, in order for the SLA to be met, due to i) the dynamic demand for the application, ii) the fact that its components are multiplexed with the components of other applications and iii) the dynamic behaviour (e.g., software stales, hardware failures, etc.) of the cloud infrastructure.

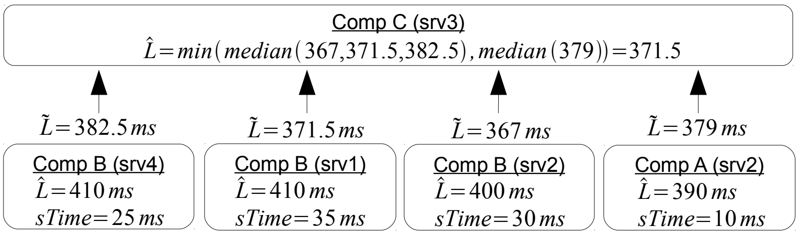


Figure 5.5: Propagation of SLA from parents to children. The child *Comp C* receives 4 SLA updates from its parents: 3 from replicas of *Comp B* and 1 from *Comp A*. The new SLA of the child component is computed following equation (5.12).

More formally, assuming that the SLA requirement is an upper bound in the response time, the response time L_j of a component j can be calculated as follows:

$$L_j = sTime_j + max_{i \in D(j)}(L_i)\,, \tag{5.10}$$

where $sTime_j$ is the service time of the j component, which is the time required by a component j to process the request locally, not accounting for the response time of the components $D(j)$ that it depends on. Therefore, the response time of a component is the sum of its service time and the response time of the slowest of the components that it depends on.

In order to meet the user requirements, each component j periodically propagates the individual *suggested* performance constraints (e.g., an upper bound $\widetilde{L_i}$ in the response time) to its dependencies $i \in D(j)$ according to the following formula:

$$\widetilde{L_i} = \widetilde{L_j} - \kappa \cdot sTime_j \; - \lambda \cdot prop_{ji}, \forall i \in D(j)\,, \tag{5.11}$$

where $prop_{ji}$ is the network delay between components j and i, while κ, λ are factors (typically 1.1) to take into account the volatility of the service time and the network delay respectively. These performance constraints are called suggested because the components $D(j)$ may receive performance constraints from other dependent components, not only from j.

When a component receives individual performance constraints from dependent components, it groups together the constraints that come from replicas of a certain component. Then, in order to compute its individual SLA constraint $\hat{L_i}$, the component i chooses the minimum value among the median SLA values of each group, i.e.,:

$$\hat{L_i} = min\{median(\Lambda_0), \ldots, median(\Lambda_n)\}\,, \tag{5.12}$$

where Λ_g is the group of the suggested performance constraints sent by the replicas of the dependent component g, while n is the number of unique dependent components (not counting the replicas). Choosing the $median()$ performance constraint instead of the $mininum()$ or the $average()$ allows the system to be more robust against unstable hosts. Each component i should satisfy that its response time meets its individual constraint within a certain *confidence bound* d, i.e.,

$$L_i \leq \hat{L}_i - d \, .$$

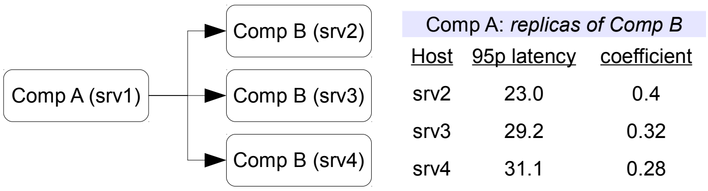
The selection of the global confidence bound implies a trade-off between the worst-case SLA-compliance and cost-efficiency. The proper value of d per application can be dynamically learnt by employing a tatonnement process for meeting the performance constraint of the application. Starting at zero, d is periodically incremented or decremented (while keeping $d \geq 0$) by a small value (e.g., 0.05) based on the percentage of time the constraint $\hat{L}_i$ is violated over a time period.

5.6 Automatic Provisioning of Cloud Resources

As explained in Section 5.3, the framework takes care of balancing the load among the available cloud resources in a fair way, by the use of autonomic migration, replication and suicide of components. However, these mechanisms might not be sufficient to ensure that the end-to-end latency is acceptable for an application owner. Essentially, each component of the distributed application needs to satisfy an individual SLA. When the application load is globally balanced, if a component is not able to process the requests fast enough, it usually means that the dedicated cloud resources are too scarce to host the application and to provide acceptable performance. A component that does not comply to its SLA is allowed to dynamically ask for more resources. If the virtual machine (VM) hosting the component is able to scale up (*vertical scaling*), the server agent will assign more resources to it (e.g., increasing the number of CPU cores, adding memory, etc.) by interacting directly with the cloud infrastructure API. If the VM is already at the maximum of its capacity, the server agent will start a new VM (*horizontal scaling*), with the minimum amount of dedicated resources. After a short period of time, some components will migrate or replicate to the new available VM. Essentially, after this load-balancing process, the component is probably able to meet its SLA. On the other hand, when a server agent realizes that the components, which it is responsible for, have enough resources to serve requests x times faster than required by their respective SLAs, the framework will decrease the dedicated resources of the VM. Thus, the adaptive provisioning of cloud resources is mainly driven by the capacity of the components to satisfy their SLA.

5.6.1 Adaptivity to Slow Servers

Each component keeps locally statistics about the latencies of its children. Every time a component sends a request to one of its dependencies, it stores the mean and the 95th percentile of its response time. With these statistics,



Comp A: *replicas of Comp B*

Host	95p latency	coefficient
srv2	23.0	0.4
srv3	29.2	0.32
srv4	31.1	0.28

Figure 5.6: *Comp A* keeps statistics about the response time of requests sent to its children locally. Based on the 95th percentile of the response time of the children, a parent computes the probability of choosing a replica of *Comp B*.

the component computes a routing coefficient for every replica of a child component (i.e., a component that it depends on) in order to dynamically choose an appropriate replica. This coefficient is the probability that the child replica will be chosen for processing of subsequent requests.

Figure 5.6 illustrates an example where there are three replicas of the child component *Comp B*. At the beginning, each child component has a coefficient of $p = 1/R = 1/3$, where R is the number of replicas, i.e., 3 in this case. Periodically, the parent component (i.e., the dependent one) updates the coefficients based on the latency of the children, as shown in pseudo-code in the Algorithm 5.1. According to the algorithm, a small value δ is added to the coefficient of the fastest component, which is subtracted from the coefficient of a randomly chosen one that is slower by more than θ. δ expresses the robustness/adaptivity trade-off of the reaction to the current component performance and it is referred to as *reactivity factor*. If the replica of *Comp B* on server *srv2* is faster than the two other replicas, then, after some time, it will have a greater coefficient, e.g., 0.4, while the coefficients of components *Comp B* on servers *srv3*, *srv4* will become 0.32, 0.28 respectively. So, on the average, *Comp B* on *srv2* will receive 40% of the requests from the parent, *Comp B* on *srv3* 32% and *Comp B* on *srv4* will only get 28%, so that the latency of the overall requests is minimized.

If one of the VMs (or the underlying physical server) is much slower than the others, then the components hosted at this slow VM will gradually receive less and less requests, and thus the server agent will scale the VM down. At some point, the VM may also be completely stopped.

When a new replica of a component shows up at a server (after a *migration* or a replication), only a small coefficient δ_0 (e.g., $\delta_0 = 0.1$) is assigned to the replica, in order not to overload it until it is properly initialized. The coefficient of the other replicas of the same component is then decreased by $\delta_0/(R-1)$. When a replica of a component disappears (after a suicide or a migration), its coefficient is equally shared among the rest of the replicas of the component.

5.7 Evaluation

The results regarding the load-balancing, the scalability and the fault-tolerance of the approach are discussed in Section 5.7.1. The effectiveness of our ap-

Algorithm 5.1 Routine for updating the forwarding coefficients of child components by a small value δ (e.g., $\delta = 0.05$)

```
Require: set of child components $S$,
  array $P$ of component coefficients,
  array $L$ of component 95th perc. of response times
Ensure: $\sum_{j=1}^{|S|} P[j] = 1$

  $fastest \leftarrow \arg\min_{j} L[j]$
  $slowComps \leftarrow \{\}$
  for all $j \in S \setminus \{fastest\}$ do
    if $L[j] - L[fastest] > \theta$ then
      $slowComps \leftarrow slowComps + \{j\}$
    end if
  end for
  $slower \leftarrow Random(slowComps)$
  $P[slower] \leftarrow P[slower] - \delta$
  $P[fastest] \leftarrow P[fastest] + \delta$
```

proach for meeting statistical SLA performance guarantees under varying request load and software/hardware failures is investigated in Section 5.7.2.

5.7.1 Scalability, High-Availability and Load-Balancing

Experimental Setup

We employ two different testbed settings: a *single-application* setup consisting of 7 servers and a *multi-application* setup consisting of 15 servers. In the former setup, the cloud resources serve 1 application and in the latter one 3 applications. The hardware specification of each server is Intel Core i7 920 @ 2.67 GHz, 8GB Ram, Linux 2.6.32-trunk-amd64. We run two databases (MySQL 5.1 [31] and Cassandra 0.5.0 [4]) as well as one generator of client requests for each application (FunkLoad [1] 1.10) on their own dedicated servers. Thus, the cloud consists of 4 and 10 servers in the *single- application* and the *multi-application* setup respectively. We assume that the components of the application may require 1 up to all servers in the cloud. The parameters employed in the utility formula are $C = 5$, $k = 5$, while $x_s^* = 25\%$.

We simulate the behavior of a typical user of the e-ticket application of Section 5.2 by performing the following actions: *1)* request the main page that contains the list of entertainment events; *2)* request the details of an event A; *3)* request the details of an event B; *4)* request again the details of the event A; *5)* login into the application and view user account; *6)* update some personal information; *7)* buy a ticket for the event A; *8)* download the corresponding ticket in PDF. A client continuously performs this list of actions over a period of 1 minute. An epoch is set to 15 seconds and an agent sends

[1] `http://funkload.nuxeo.org/`

gossip messages every 5 seconds. Moreover, the default routing policy is the random-based policy.

We consider two different placements of the components:

- A **static** approach where each component is assigned to a server by the system administrator.
- A **dynamic** approach where all components are started on a single server and dynamically migrate / replicate / stop according to the load or the hardware failures.

Results

Dynamic vs Static Replica Placement First, we employ the *single-application* experimental setup to compare our approach with *static* placements of the components, where we consider two cases: *i)* each different component is hosted at a different dedicated server; *ii)* full replication, where every component is hosted at every server. The response time of the 95% percentile of the requests is depicted in Figure 5.7. In the static placement (i), where a component runs on its own server, the response time is lower bounded by that of the slowest component (in our case, the service for generating PDF tickets). Thus, the response time increases exponentially when the server hosting this component is overloaded. In the case of full replication [static placement (ii)], the requests are balanced among all servers, keeping the latency relatively low, even when the amount of concurrent users is significant. In the dynamic placement approach, all components are hosted at a single server at startup: then, when the load increases, a busy component is allowed to replicate, and unpopular components may replicate to a less busy server. Our economic approach achieves better performance than full replication, because the total amount of CPU available in the cloud is used in an adaptive manner by the components: processing intensive (or "heavy") components migrate to the least loaded servers and heavily-used components are assigned more resources than others. Therefore, the cloud resources are shared according to the processing needs of components and no cloud resources are wasted by over-provisioning.

Also, as the cloud resources are properly utilized by the economic approach, the application throughput (i.e., the number of request served per second) that it achieves outperforms static placements, as depicted in Figure 5.8.

Scalability Having established the effectiveness of our dynamic component placement approach over static ones, we next investigate the resulting scalability in the cloud in the *multi-application* experimental setup. We assume that all 10 servers reside at 1 datacenter. We gradually increase the number of concurrent users from 150 to 1500. The service requests are equally shared among applications and randomly routed among the replicas of a component. As depicted in Figure 5.9, the response time per application increases linearly to the load and the resources of the cloud are shared among the components of different applications in a fair way. The experiment is repeated 5 times and mean values and confidence intervals are presented.

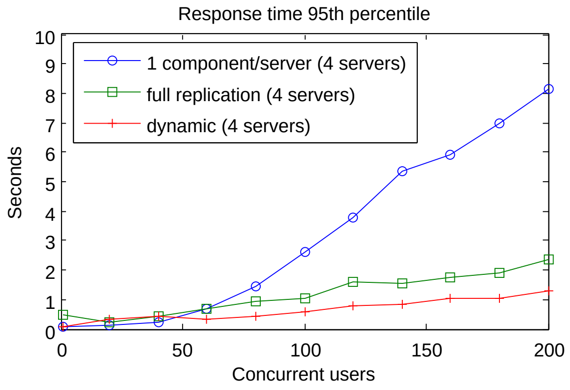


Figure 5.7: Response time of different placement approaches

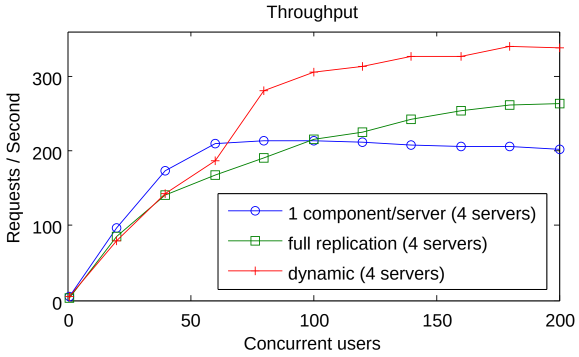


Figure 5.8: Throughput compared with different placement approaches

High-Availability Next, we show that our approach is highly resilient to hardware failures. To this end, we employ the *single-application* experimental setup. We assume that each component has 2 replicas (i.e., 2 instances) that reside at separate servers. 10 concurrent clients continuously send requests for 1 minute. After 30 seconds, one random server between those hosting the replicas of a component fails. As illustrated in Figure 5.10, the percentage of requests that were not satisfied is 0.34% in this case. The failures correspond to requests already sent to the failed replica. If both servers hosting the replicas of a component fail at the same time, 3.58% of the requests are lost. Due to the gossiping protocol, the remaining servers quickly detect the failure, start the failed components locally and broadcast the updated routing entries for them.

Adaptation to New Cloud Resources In this experiment, we investigate the adaptability of our dynamic placement approach when new resources are added to the cloud. We employ the *single-application* experimental setup, but the number of available servers in the cloud ranges from 1 to 10. The application is concurrently accessed by 1500 users, while service requests are equally shared among the replicated instances of a particular component. As

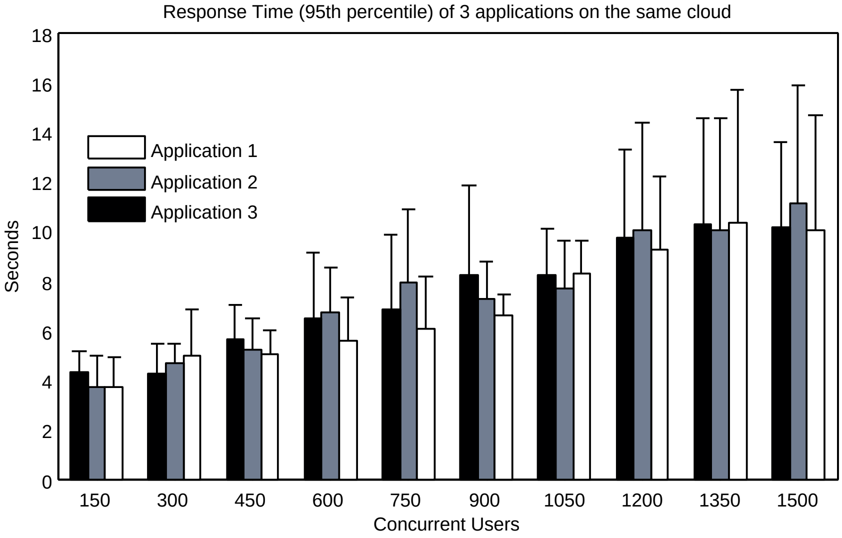


Figure 5.9: 95% percentile response times for 3 different applications as load increases

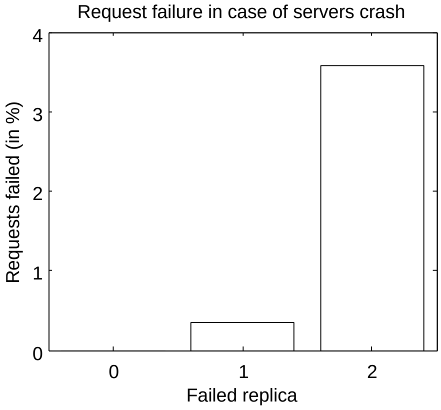


Figure 5.10: Request failure percentages when 0, 1 and 2 replicas (out of 2) crash

observed by Figure 5.11, our dynamic placement approach fully exploits the new server resources that are added to the cloud, as the application response time decreases, while the service throughput increases. The experiment is repeated 5 times and mean values and confidence intervals are presented.

Evaluation of Routing Policies When multiple instances of a particular component are available, the requests have to be split among the several instances of the requested component to efficiently balance the load. One approach could be that the requests are equally shared (i.e., at random) among the instances of the requested component. However, this approach does not take into account neither the network delay among the service hosting the requesting and the requested components, nor the load at the servers hosting

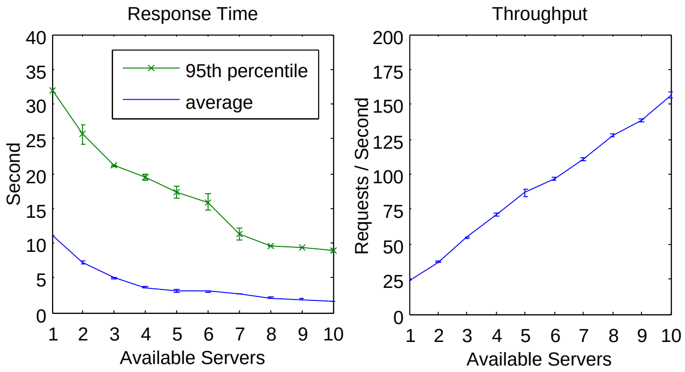


Figure 5.11: Response time (left) and throughput (right), when new cloud resources are added

the instances of the requested components. To this end, we experimentally investigate the three other approaches that were described in Section 5.3.3.

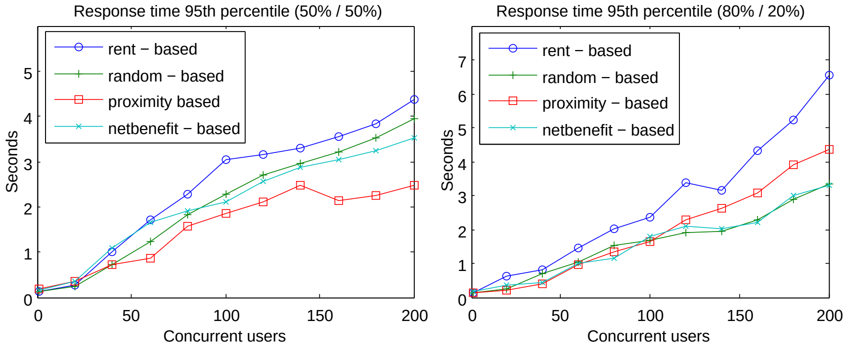


Figure 5.12: Response time when requests between datacenters are: i) 50%-50% (left), ii) 80%-20% (right)

In this case, we employ the *single-application* setup, but the 4 servers of the cloud are located in 2 datacenters (2 servers per datacenter). The round-trip time between the datacenters is 50 ms and the minimum availability (i.e., number of replica per component) is set to 2.

First, the application requests are evenly split between the two datacenters (50% to datacenter 1, 50% to datacenter 2). As depicted in Figure 5.7.1(left), the proximity-based routing policy achieves the lowest response time, as it saves the delay for transmitting requests between the datacenters. However, it may result in an unbalanced server usage if the traffic is not balanced between the datacenters, as illustrated in Figure 5.7.1(right). The rent-based policy (as well as the net benefit one) may suffer from the fact that the rents of servers may not always be up-to-date due to the gossiping algorithm. The net benefit routing policy performs at least as good as the random one both for balanced and for unbalanced load between datacenters, as depicted in Figure 5.7.1, yet at a higher computational cost at runtime.

5.7.2 SLA Performance Guarantees

Experimental Setup

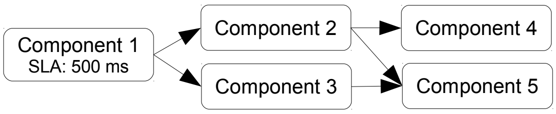


Figure 5.13: Architecture of a test application composed of 5 components

In our evaluation, we consider an application composed of 5 different components, as depicted on Figure 5.13. The results presented in this section refer to an application that is mostly CPU-intensive, hence $w_c >> w_m, w_n, w_d$ for every component. However, we have also conducted experiments with different types of applications (I/O intensive, CPU intensive, a mix of both, etc.) and with different component workflows (fully parallel, fully sequential, a mix of parallel/sequential with several numbers of tiers) with similar conclusions on the effectiveness of our approach. At startup, all components are started on a single 1-core server. The minimum number of replicas per component for ensuring fault-tolerance is set to 2.

The underlying cloud infrastructure is composed of 8 8-cores servers (Intel Core i7 920 @ 2.67 GHz, 8GB Ram, Linux 2.6.32-trunk-amd64). The components interact with the cloud infrastructure through an API that allows asking for more resources (adding cores to a server, starting a new server) or less resources (remove cores from a server, stopping a server). Although each server CPU has 8 cores, we only allow a server to employ 2, 3 or 6 cores at maximum in our evaluation, in order to force the earlier provisioning of a new server. The servers reside on a 1Gbps switched Ethernet LAN.

We have set the SLA (by means of an upper bound in the response time) of the first component of the application (i.e., "Component 1" on Figure 5.13) to be 500 ms, while no confidence bound (i.e., $d = 0$) was considered. The parameters employed in the utility formula are $C = 5$, $k = 5$, while $x_s^* = 25\%$.

Results

Adaptation to Varying Load In this experiment, we investigate the reactivity of our framework to quickly increasing or shrinking load of application requests. The initial load is assumed to be 5 requests per second. At the 8-th minute of the experiment, we start increasing the load every minute by 5 requests per second until the total traffic reaches 60 requests per second. Then, we keep the request load constant for 15 minutes. Afterwards, we start decreasing the load every minute by 5 requests per second until the initial load is reached. We allow a maximum number of 3 cores to be employed per server.

We compare the performance (in terms of response time) of our dynamic approach with that of a static one under the same load conditions. In the

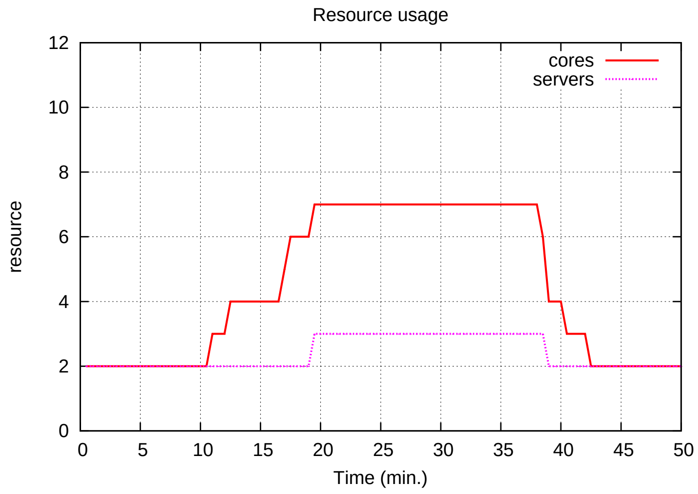


Figure 5.14: *Scarce*: Resources used by the application over time for varying request load

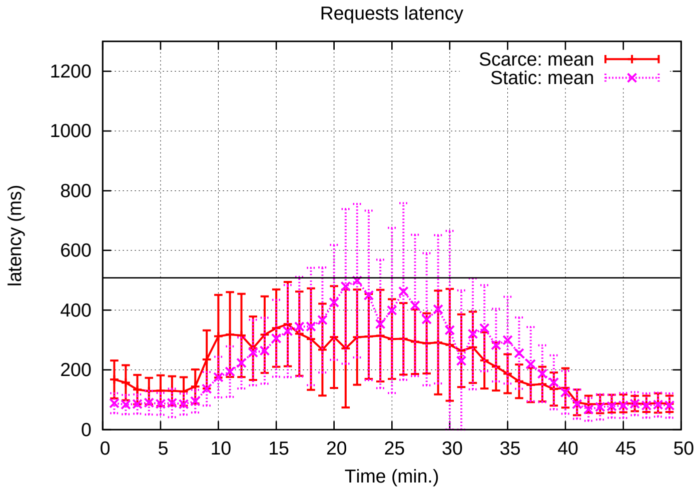


Figure 5.15: Mean response times of the application (SLA: 500 ms) as perceived by remote clients under the adaptive approach (*Scarce*) and the static setup

static setup, the total amount of cloud resources allocated to the application remains constant and equal to 2 servers with 2 cores each. During the total time of the experiment (60 minutes), our dynamic approach has employed for the application components 4 cores on the average from the cloud. For a fair comparison regarding the performance, we also employed a total of 4 cores for the application in the static setup.

As depicted in Figure 5.14, our framework reacts appropriately to the increasing amount of requests by asking for more cloud resources in order to satisfy the SLA. Once the additional resources are no longer required for SLA compliance (i.e., the 95th percentile response time of every component in the server is x times faster than required), then the framework releases them for reducing the costs. Clearly, in the long run, the overall cost for the static setup would be much higher, as the user would constantly pay for 4 cores even in low load periods (where 2 cores can satisfy the SLA). Our framework uses the minimum required resources to serve the application within the SLA requirements by fully leveraging the elasticity of today's cloud infrastructures.

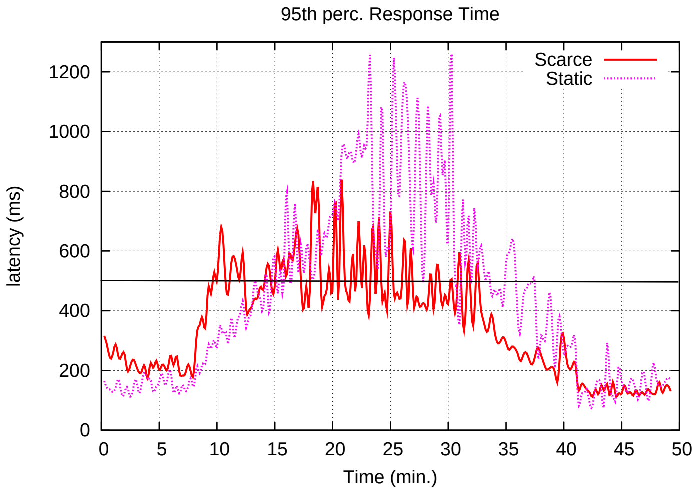


Figure 5.16: 95th percentile response times of the application (SLA: 500 ms) as perceived by remote clients under *Scarce* and the static setup

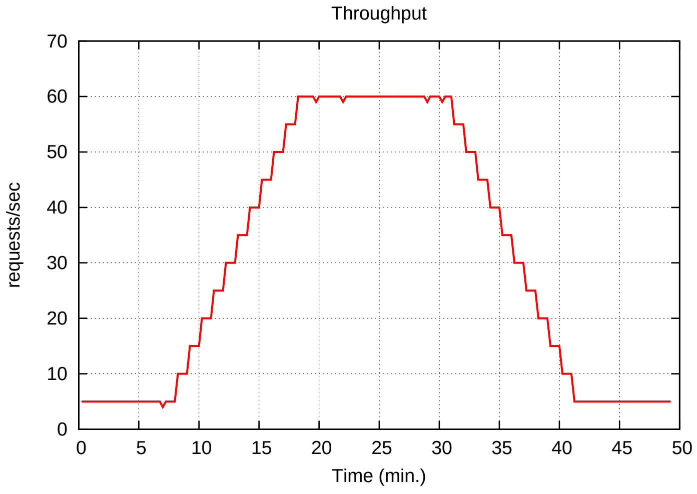


Figure 5.17: Throughput of the application during the varying load experiments

Also, thanks to the adaptivity of our framework, the maximum request load for the application (60 requests per second) can be sustained, while keeping the average response time under 500 ms, as shown in Figure 5.15. The static approach is also able to serve the maximum request load, but the average response time is greater and significantly varies with time. Also, our adaptive framework achieves much lower 95th percentile of the response time than that of the static approach, as depicted in Figure 5.16. The framework only reacts when the 95th percentile of the response time reaches 500 ms. Until minute 13, the static setup has more resources (i.e., 2 servers with 2 cores) compared to *Scarce*, and therefore performs better. After minute 20, *Scarce* has allocated the needed resources to the application to meet the SLA and clearly outperforms the static setup.

However, the confidence bound d should be properly selected, according to the application tolerance to the QoS violations. There is a clear *trade-off* between worst-case SLA-compliance and cost-efficiency in the selection of the confidence bound. As shown in Figure 5.19, if the application is assumed to be inelastic and the confidence bound is selected as $d = 60\% \cdot response\_time$, then our adaptive approach would allow almost no SLA violations, as opposed to the static setup.

As soon as the number of requests per second sent to the application increases, the service time as well as the response time of each component are impacted. Recall that each component periodically sends a suggested SLA constraint update to its child components. As the suggested SLA update (given by equation (5.11)) depends on the component service time, the application load has a direct effect on the derived SLA constraint of each component. This effect is depicted in Figure 5.18, where the SLA constraints for the components are getting stricter with the growing application load.

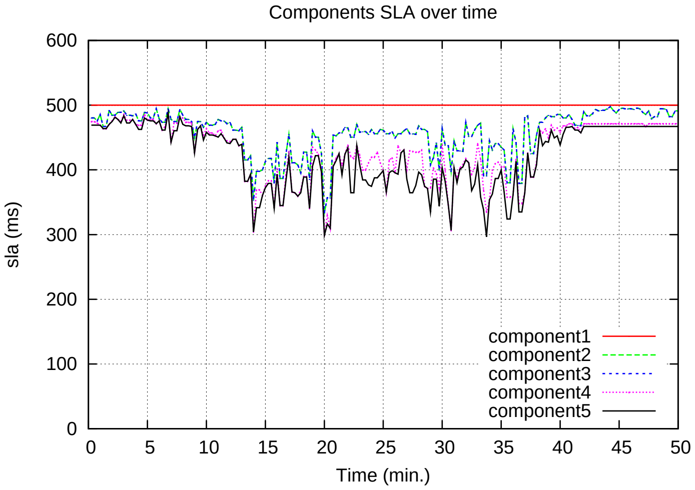


Figure 5.18: Computed SLA constraints of the components hosted at a server

Adaptation to Slow Servers A recurring issue with cloud infrastructures is that the user has no control over the performance of the rented resources. As

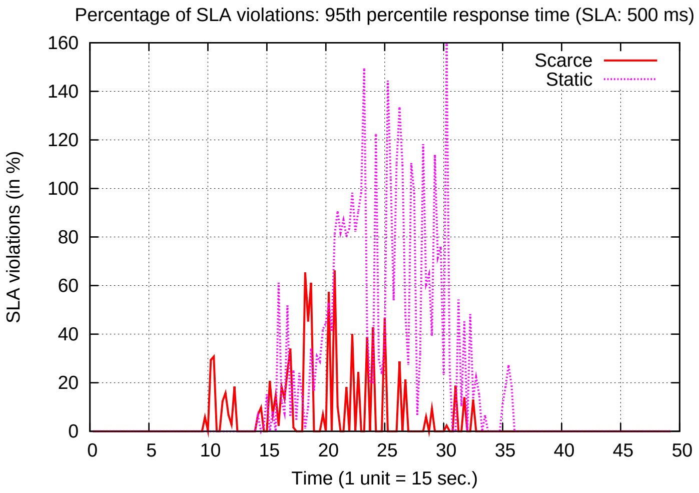


Figure 5.19: Percentage of SLA violations from *Scarce* and the static approach when the 95th percentile response time should stay under 500 ms

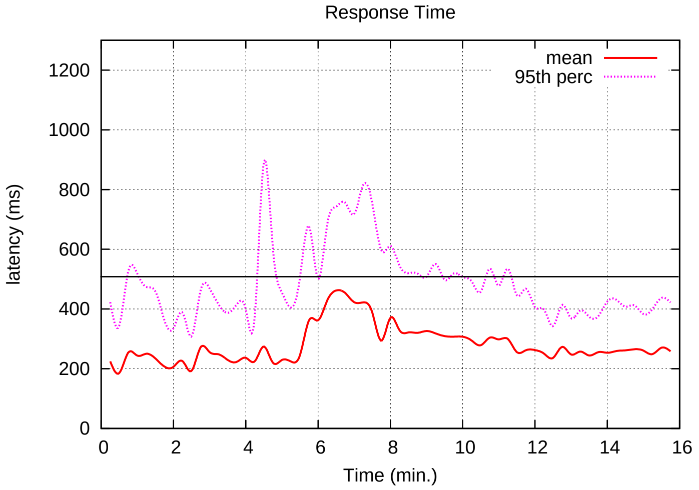


Figure 5.20: Mean and 95th percentile response times of the application (SLA: 500 ms) as perceived by remote clients in case of a "wonky" server

a physical server is shared by several virtual machines (VM), a VM might intensively use the I/O subsystem, and may therefore degrade the performance for the other VMs collocated on the same physical server. In addition, a physical server, which has an unreliable hardware component or a non-optimized operating system setup, will also have poorer performance, and will therefore negatively impact the application end-to-end latency. Our framework is able to detect slower servers and to discard them transparently to the application. In this case, the maximum number of cores allowed to be employed per server is 2.

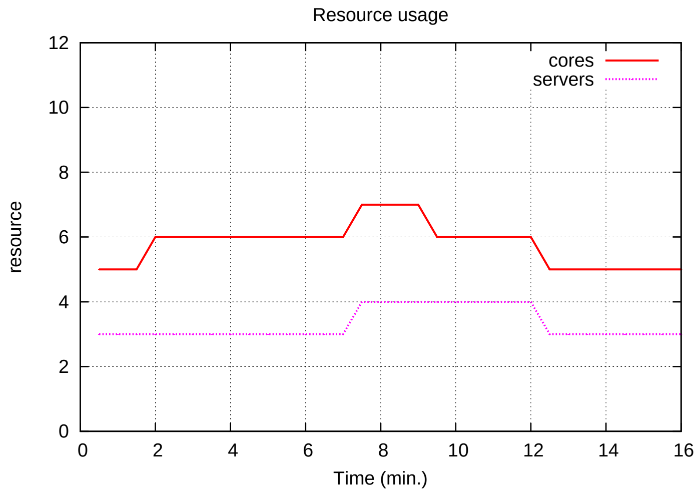


Figure 5.21: Resources used by the application over time in case of a "wonky" server

Here, the request load is assumed to be 25 requests per second to the entry component of the application. After 4 minutes, one of the server starts to be slower and every component hosted at it serves requests with a delay of 200ms. All components that are not hosted at the "wonky" server detect this slowdown, adapt the coefficients of their dependencies accordingly and the traffic is slowly redirected to faster components. Two minutes later (at minute 6), a new server is started, because some components are no more able to honor their SLA due to the redirected traffic. At minute 11, the wonky server is removed from the cloud by the framework as it receives only a negligible amount of requests. As shown in Figure 5.20, our approach quickly adapts to the situation and renders the response time of the application again compliant to the SLA. The dynamic resource allocation for the application in this scenario is depicted in Figure 5.21.

Scalability and Stability In this experiment, the rate of requests for the application increases every minute by 5 requests/second until reaching the load of 150 requests/second. Each server is allowed to employ up to 6 cores. In Figure 5.22, the 95th percentile of the response time quickly stabilizes close to the SLA constraint after the request rate stops increasing and becomes constant. Finally, as shown in Figure 5.24, the global amount of physical resources employed follows the trend of the request load (depicted in Figure 5.23).

Discussion

Simple approaches proposed by cloud providers such as Amazon allow a customer to set rules to automatically add or remove resources when a metric (e.g., CPU usage) goes above or below a threshold H and L respectively. However, even if the CPU usage higher than H, the performance constraint

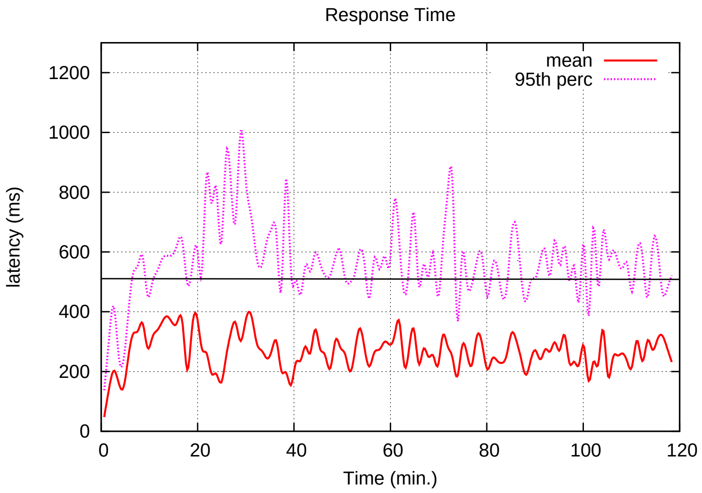


Figure 5.22: Mean and 95th percentile response times of the application (SLA: 500ms) as perceived by remote clients in the scalability experiment

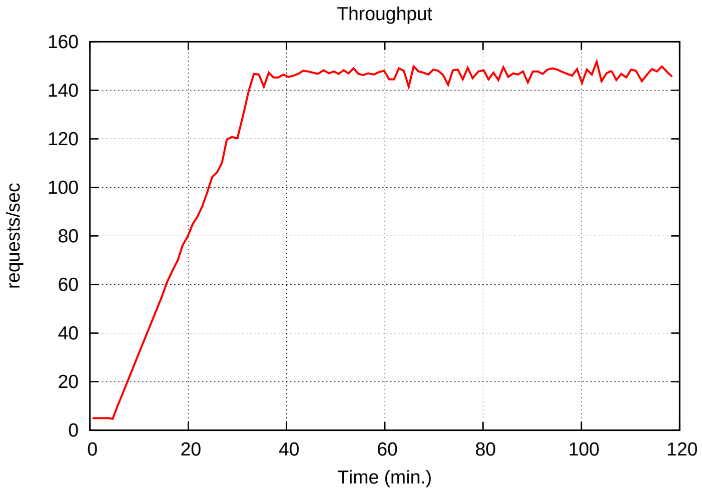


Figure 5.23: *Scarce*: Throughput of the application during the scalability experiment

per component may be met, or vice versa. The employment of fine-grained metrics, such as 95th percentile of response time per component (server metrics are not enough), is required to use the minimum amount of resources for a given SLA. Moreover, *Scarce* efficiently multiplexes components at servers based on component migration and replication.

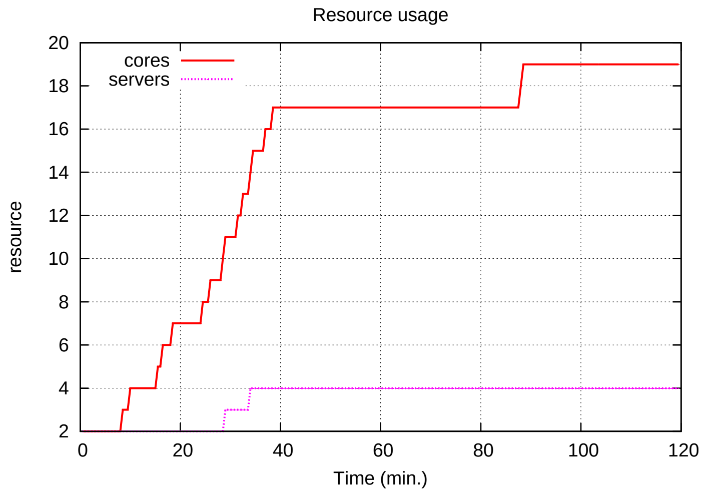


Figure 5.24: Resources used by the application over time during the scalability experiment

5.8 Related Work

There is significant related work in the area of economic approaches for resource management in distributed computing. In [157], an approach is proposed for the utilization of idle computational resources in an heterogeneous cluster. Agents assign computational tasks to servers, given the budget constrain for each task, and compete for CPU time in sealed-bid second-price auction held by the latter. In a similar setting, Popcorn approach [132] employs a first-price sealed-bid auction model.

Cougaar distributed multi-agent system [101] has an adaptivity engine which monitors load by employing periodic "health-check" messages. An elected agent operates as load balancer and determines the appropriate node for each agent that must be relocated based on runtime performance metrics, e.g., message traffic and memory consumption. Also, a coordinator component determines potential failure of agents and restarts them. However, cost-effectiveness is not among the objectives of Cougaar, and moreover our approach is more lightweight in terms of communication overhead.

In [69], a virtual currency (called Egg) is used for expressing a user's willingness to pay as well as a provider's bid for a accepting the job, and finally is given to the winning provider as compensation for job execution. The central Egg entity informs all candidate providers about the new job and acquires responses (opportunity cost estimations for accepting the job). However, the approach in [69] is centralized and it does not provide availability guarantees.

In [125], applications trade computing capacity in a free market, which is centrally hosted, and are then automatically activated in virtual machines on the traded nodes on-call of traffic spikes. The applications are responsible for declaring their required number of nodes at each round based on usage statistics and allocate their statically guaranteed resources or more based on their willingness to pay and the equilibrium price; this is the highest price at which the demand saturates the cluster capacity. However, [125] does not deal with availability guarantees, as opposed to our approach. Also, our approach accommodates traffic spikes in a prioritized way per application without requiring the determination of the equilibrium price.

Pautasso *et al.* propose in [127] an autonomic controller for the JOpera distributed service composition engine over a cluster. The autonomic controller starts and stops navigation (i.e., scheduler) and dispatcher (i.e., execution and composition) threads based on several load-balancing policies that depend on the size of their respective processing queues. However, proper thread placement in the cluster and communication overhead among threads are not considered in [127].

Also, SLA provisioning for web services [76] or multi-tier web applications [156, 119] has been studied. [156, 119] adapt the behavior of the underlying resources based on the capacity of an application to honor an SLA. However, monitoring of SLA compliance in [76, 156, 119] may require the involvement of third-parties or centralized services. A decentralized approach for SLA provisioning in grids based on service migration between containers is proposed in [133]. However, this approach does not dynamically vary the total amount of allocated resources to the application according to the load and the potential software/hardware failures in the cloud, as opposed to our work.

A bio-networking approach was proposed in [158], where services are provided by autonomous agents that implement basic biological behaviors of swarms of bees and ant colonies such as replication, migration, or death. To survive in the network environment, an agent obtains “energy” by providing a service to the users. However, [158] does not consider the problem of satisfying performance constraints.

Moreover, several implementation frameworks exist towards reliable SOA-based applications: [117] is a mechanism for specifying fault tolerant web service compositions, [83] is a virtual communication layer for transparent service replication, and [139] is a framework for the active replication of services across sites. These frameworks do not consider dynamic adaptation to changing conditions, such as load spikes, or do not provide guarantees for geographical diversity of replicas.

5.9 Conclusion

We proposed an economic, lightweight approach for dynamic accommodation of load spikes and failures for composite web services deployed in clouds, so as to satisfy performance and availability guarantees. We derive performance constraints per component and we scale up existing VMs or create new whenever they are not met. Application components act as individual optimizers and autonomously replicate, migrate across VMs or terminate based on their economic fitness. Their resource inter-dependencies are implicitly taken into account by means of server rent prices. The requests are routed across components based on their respective prior performance. Our approach also detects unstable cloud resources and reacts accordingly, so as to minimize the end-to-end application response time. As a future work, we intend to explore our economic paradigm for autonomic resource management in the context of multiple competitive or cooperative cloud providers.

Chapter 6

Federation of Cloud Storage

A growing amount of data is produced daily resulting in an increased demand for storage solutions. While cloud storage providers offer a virtually infinite storage capacity to their customers, data owners seek geographical and provider diversity in data placement, in order to avoid vendor lock-in and to increase availability and durability. Moreover, the cloud storage providers have chosen their pricing policies very carefully, so as to attract a precise class of consumers and according to their specific resource constraints. Also, depending on the data access pattern, a certain cloud provider may be cheaper than another one. In this chapter, we introduce *Scalia*, a cloud storage brokerage solution that continuously adapts the placement of data based on its access pattern and subject to optimizations objectives, such as storage costs. *Scalia* is implemented based on a scalable architecture that cleverly considers the re-positioning of only selected objects that may significantly lower the storage cost. By an extensive series of simulation experiments, we prove the cost-effectiveness of *Scalia* against static placements and its proximity to the ideal dynamic placement in various scenarios of data access patterns, of cloud resources available and of pricing policies.

6.1 Introduction

Cloud providers are offering efficient on-demand storage solutions that can virtually scale indefinitely. Many public cloud storage providers are already available in the market, such as Amazon S3 [2], Google Storage [16], Microsoft Azure [27] or RackSpace CloudFiles [42] and one may expect new providers to appear in the coming years. The offers in terms of pricing among providers vary significantly and may change over time to adapt to the market. Choosing the best-suited or cheapest provider for your data implies knowing in advance the access pattern to the data. Data that is rarely accessed should be stored at a cloud provider mainly with a low storage price, regardless of its access prices. On the other hand, a very popular data may be hosted on a provider with attractive price for the outgoing bandwidth. In most cases, it is difficult to know in advance the access pattern of a data item, and therefore one needs an adaptive solution to choose the most cost-efficient provider.

However, finding a suitable provider based on the access pattern of the data is not enough. A provider may end its business or suddenly increase its pricing policy. There exist many other technical as well as non-technical (e.g., boycotting a provider) reasons a user may want to change its provider. Therefore, in order to safely host its data and minimize the impact of the migration to a new provider, a user needs to proactively avoid vendor lock-in (i.e., being dependent on a specific service vendor with substantial switching costs) and ensure high durability and availability by geographic diversification of the data placement (e.g., the recent Amazon outage [1] reminds us not to put all eggs in one basket).

The authors of [57] clearly underline the advantages of splitting a data object (e.g., a file) into chunks and storing them across several storage providers, in order to reduce costs and avoid vendor lock-in. However, a more adaptive approach is required to cope with dynamically changing conditions, such as varying data access patterns, evolving pricing policies, new providers arrival, as well as providers' bankruptcy. Moreover, different data access patterns result in different optimal sets of providers in terms of charging.

In this chapter, we introduce *Scalia*, a system that continuously adapts the placement of data among several storage providers subject to optimization objectives, such cost minimization. Our system combines the following unique and novel characteristics:

1. Adaptive data placement based on the real-time data access patterns, so as to minimize the price that the data owner has to pay to the cloud storage providers given a set of customer rules, e.g., availability, durability, etc. Other optimization goals for data placement are also conceivable, such as a) maintaining a certain monthly budget by relaxing some constraints, such as lock-in or availability, or b) minimizing query latency by promoting the most high-performing providers.

2. Compliance with the rules set by customers for data, such as data durability, data availability and level of vendor lock-in.

3. Orchestration of a non-static set of public cloud and corporate-owned private storage resources.

4. A robust distributed architecture for its implementation that is able to handle a large number of objects stored, which are accessed by a large number potential users.

The remainder of this chapter is organized as follows: In Section 6.2, we discuss the problems of vendor lock-in and paying unfairly high prices when fixed sets of cloud storage providers are employed. In Section 6.3, we describe our *Scalia* brokerage architecture, the adaptive data placement mechanism, the data caching layer, the metadata storage layer and *Scalia* actions in data read/write operations. In Section 6.4, we assess the effectiveness of our approach for cost-effective data placement. In Section 6.5, we present some related work, and finally, in Section 6.6, we provide our concluding remarks.

[1] `http://aws.amazon.com/message/65648/`

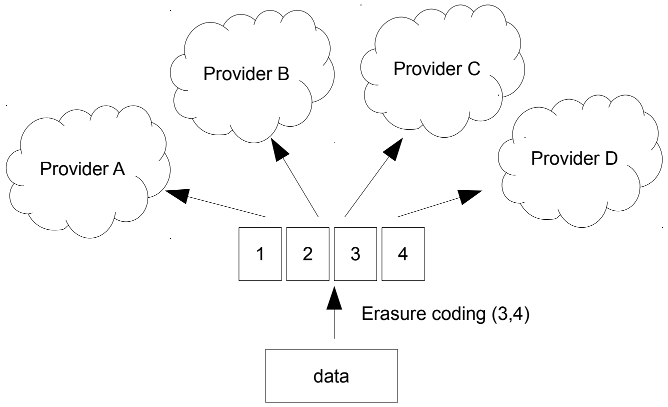


Figure 6.1: Erasure coding (m, n): any m-subset of the n chunks contains a complete copy of the data

6.2 Motivation

6.2.1 Avoiding Vendor Lock-in

Erasure Coding

In order to avoid vendor lock-in, data has to be hosted by multiple storage providers. However, despite being simple and reactive, storing full replicas of the same data is too costly [159, 136]. With the aid of erasure coding (m, n) [82], a data can be split into n chunks ($n > m$), where any m-subset is sufficient to reconstruct a complete copy of the data. The rate $r = \frac{m}{n} < 1$ of an erasure code is the fraction of chunks required to rebuild the original data. The disk space needed to store an r-encoded object increases by a factor of $\frac{1}{r}$. In Figure 6.1, the original data can be rebuilt with the chunks stored at any 3 of the 4 cloud providers. For example, RAID 1 (mirroring without parity or striping) can be achieved by setting $m = 1$, while RAID 5 (block-level striping with distributed parity) can be described by $(m = k, n = k + 1)$, where $k \geq 3$.

Redundant striping presents several advantages. First, it allows to tolerate up to $n - m$ provider outages, hence greatly improving the durability as well as the availability of the stored data. The user may also choose how to recover from a provider failure. One might decide to reconstruct the missing chunks from the other providers and store them to new providers, or on the other hand, one might decide to ignore the failure and wait for the provider to recover. Second, striping provides a finer granularity than full replication, which permits to read from the cheapest provider or to move a restricted number of chunks to a cheaper provider. Also, it gives a better control on the cost by allowing to store and serve data from public providers as well as private storage facilities.

Table 6.1: Example of storage rules

name	*durability*	*availability*	*zones*	*lock-in*
rule 1	99.99999	99.99	EU, US	0.3
rule 2	99.999	99.99	EU	1
rule 3	99.99	99.99	all	0.2

6.2.2 Paying a Fair Price

Given customer (i.e., data owner/producer) requirements (possibly differentiated per data item), such as data durability, data availability or independence from cloud providers to avoid vendors lock-in, it then becomes a non-trivial task to find the cloud storage provider(s) or combinations of cloud storage providers that offer the best price to store users' data. To make things worse, the ratio of read/write operations of a data object over a period of time (i.e., the data access pattern) affects the resulting charging for the customer, as providers implicitly promote certain access patterns with their pricing policies. *Scalia* provides an engine that optimizes the placements of data chunks following the rules set by the data owner, while also taking into account the access patterns of the data in order to compute the cheapest provider set. A default rule, rules per data object classes or rules per data object can be defined in *Scalia* (e.g., using an API or a Web interface), so as to specify the availability, the durability, the geographical zone(s) and the lock-in factor of the data, as described in Table 6.1. The lock-in factor $obj[lockin] \in (0,\ 1]$ of a data object obj is defined as:

$$obj[lockin] = \frac{1}{N_{obj}}, \tag{6.1}$$

where N_{obj} is the number of minimum distinct providers where the data object obj will be stored.

Table 6.2: Example of provider sets (c.f Table 6.3 for abbreviations)

name	*durability*	*availability*	*zones*
S3(h)	99.999999999	99.9	EU, US, APAC
S3(l)	99.99	99.9	EU, US, APAC
RS	99.9999	99.9	US
Azu	99.9999	99.9	US
Ggl	99.9999	99.9	US

Table 6.3: Providers' abbreviations.

abbreviation	*description*
S3(h)	Amazon S3 with high durability
S3(l)	Amazon S3 with low durability
RS	RackSpace CloudFiles
Azu	Microsoft Azure
Ggl	Google Storage

Given the users' rules, the engine stores the user data at the cheapest provider set among the complete range of possible alternatives, and continuously adapts the data placement to match the data access pattern. For example, a user looking to store non-critical and ephemeral data will not be interested in avoiding vendor lock-in or storing its data to a high durability provider. On the other hand, if one wants to store critical data over a long period of time, vendor lock-in as well as durability become serious issues. Cold data may be stored at providers offering the cheapest storage price, disregarding the price of bandwidth or operations, while popular data should be stored to providers showing interesting prices regarding outgoing bandwidth. By only specifying simple rules, a user should be able to always pay a fair price, corresponding exactly to his real needs.

Table 6.4: Example of providers prices in USD per GB for storage, bandwidth in and out, or in USD per 1000 requests for the operations (c.f Table 6.3 for abbreviations).

name	*storage*	*bdw in*	*bdw out*	*ops*
S3(h)	0.14	0.1	0.15	0.01
S3(l)	0.093	0.1	0.15	0.01
RS	0.15	0.08	0.18	0.0
Azu	0.15	0.1	0.15	0.01
Ggl	0.17	0.1	0.15	0.01

6.3 Scalia: Multi-Cloud Storage

In this section, we describe *Scalia* in detail and present its complete architecture, which enables to aggregate public cloud storage providers and private storage resources. *Scalia* can run directly at the customer premises as an integrated hardware and software solution (i.e., an appliance) or can be deployed as a hosted service across several datacenters, putting the emphasis on providing a scalable and highly-available architecture with no single point of failure, able to guarantee higher availability than the storage providers. In the first deployment model, the appliance is located directly in the customer's data center, with the advantage of not introducing additional network latency, not having to pay any extra service fee and not being dependent on the availability of the hosted service. On the other hand, when *Scalia* is accessed as hosted service, a customer does not need to install any additional hardware or software and will pay only service fees. The hosted service can be operated by an independent broker for multiple customers.

In Figure 6.2, for simplicity, we consider *Scalia* as a hosted service in a setup consisting of only a pair of datacenters. A client can send requests indifferently to each datacenter. The *Scalia* brokerage system consists of three layers: a layer of stateless engines, a caching layer and a database layer. The engines provide an Amazon S3-like interface, where the users can put, get, list and delete their data using a key-value data model. The engines are responsible for computing the best provider set according to the user requirements, for maintaining the cost-effective data placement using the access history of the data, for splitting and storing the chunks at the most suitable providers,

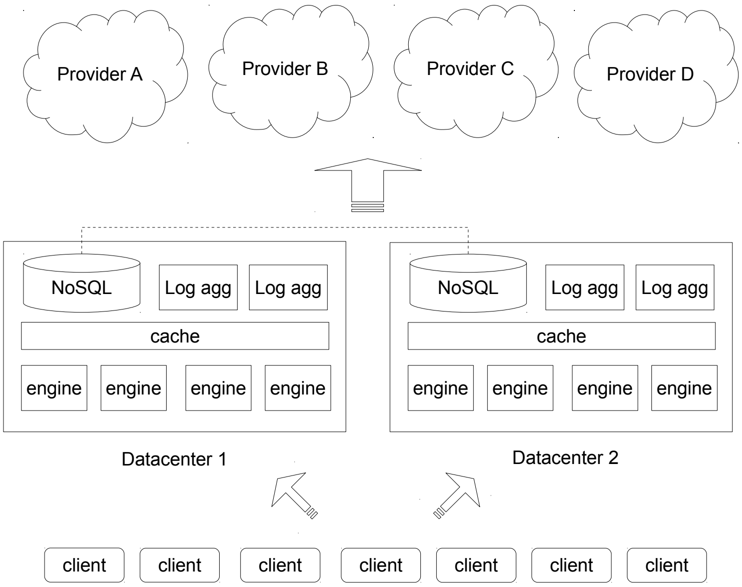


Figure 6.2: Multi datacenter architecture

for reconstructing the data from the chunks and finally for deleting the data. Each engine works independently and does not keep a state. This allows this layer to scale linearly by just adding new engine components. The caching layer is not mandatory, yet if employed, it greatly improves the performance for read operations of popular data and reduces the corresponding costs for data fetching. The database layer is responsible for hosting the metadata of the data stored in the remote storage providers, and to store their access statistics.

6.3.1 Engine Layer

The engine acts as a proxy between the client and the cloud storage providers, offering a unified API to all providers, including data storage to private resources. Mainly, it is responsible for storing the chunks of data to the optimal providers, and serving the data either directly from the cache or by reconstructing it using the chunks stored at the remote providers.

The engine also takes care of maintaining the optimality of the chunk placement of an object obj, by periodically recomputing the best provider set using the data access statistics of the last $|D_{obj}|$ sampling periods, where $D_{obj} \subset H_{obj}$ is referred to as *decision period* and corresponds to the period of historical access statistics used to compute the optimal chunks placement. The access statistics of a data object obj are kept in the history H_{obj}. The sampling period s is a time period where the statistics per object are collected and aggregated, typically 1 hour. Knowing the recent access history of a data permits to precisely adjust the set of providers, as we can reasonably suppose that the access pattern of the data in the near future will be similar to the current. Choosing a large decision period allows to predict the access pattern farer in the future, and thus permits to make better placement choices in the long run. However, imagine that the chunks of a data object were placed based on the assumption that the object would be stored for at least 6 months, and the object was in fact deleted after 1 week. The chosen placement would have been probably wrong, resulting in higher costs for the end user. Thus, the decision period D_{obj} has to be dynamically adjusted as it depends on the lifetime of the object, the burstiness of its access pattern and the resulting economic impact of the latter.

In practice, it is determined based on a dichotomic search between 0 and $min(TTL_{obj}, H_{obj})$, where TTL_{obj} is the time left to live of the data object *obj*, as described below. When a periodic optimization procedure begins (as will be described in Section 6.3.1), historical access statistics of length $\frac{D_{obj}}{2}$, D_{obj}, and $2 * D_{obj}$ are considered in parallel (i.e., coupling) when computing the best set of providers using Algorithm 6.1. D_{obj} is then updated to the decision period based on which the cheapest set of providers is found among the three best sets. This approach for updating D_{obj} is applied every T optimization procedures. Initially, $T = 1$ and whenever D_{obj} is found to be adequate, T is doubled, otherwise, T is reset to the initial value, i.e., $T = 1$. The maximum value of T can be considered to be a period of weeks. We consider here two approaches to determine TTL_{obj}:

- an indication of the object lifetime can be provided by the end user at write time, allowing *Scalia* to make the optimal choices;
- otherwise, *Scalia* employs statistics collected from all data objects to find out the most probable lifetime of a certain data item, as will be explained in the next subsection;

Classification of Objects

An object belongs to a class of objects determined by its metadata such as size or MIME type. The class of an object $C(obj)$ is derived using a simple hash of relevant metadata:

$$C(obj) = MD5(obj[mime] \mid discretize(obj[size]))$$

where *discretize*() is a function which rounds a number to a close integer (e.g., the size of an object is rounded up to the closest megabyte).

For every class of object, *Scalia* collects statistics regarding the resources used (i.e., bandwidth in and out, operations, deletion time, . . .) and computes the lifetime distribution of the class, in order to dynamically assign a satisfying value for the decision period D_{obj} and to predict the lifetime of a new object at the time of insertion. As shown in Figure 6.3, given the deletion time of the objects of a certain class (left), one can compute the most probable time left-to-live for an object (right). For example, at insertion time, the lifetime of an object of that class is expected to be 3.25 hours, while a 2 hours old object is expected to live for 1.55 hour more. The lifetime distribution of the classes of objects stabilizes after a training phase, and thus does not incur

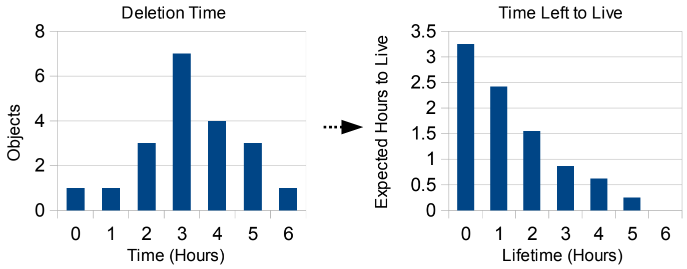


Figure 6.3: Time left to live for a class of objects, as computed by the statistics. The class contains 20 objects, whose lifetime varies from 0 to 6 hours.

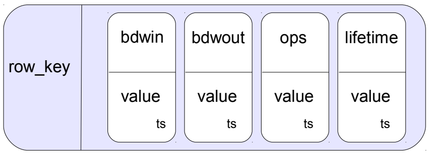


Figure 6.4: Statistics are used to improve the first placement of an object

extra computing costs. The statistics and distributions of the classes of objects are periodically refreshed using map-reduce jobs in the database layer.

Placement Algorithm

The time is divided into sampling periods. In current public cloud storage system, this period usually corresponds to 1 hour. For a sampling period s_i at time i, statistics of a data object obj are collected, such as the used storage $s_i[storage]$, the incoming bandwidth $s_i[bwdin]$, the outgoing bandwidth $s_i[bwdout]$ as well as the number of operations $s_i[ops]$. Let $H(obj) = \{s_{t-0}, s_{t-1}, s_{t-2}, \ldots, s_{t-|H|}\}$ be the list of access history statistics of the data object at time t.

At insertion time, a data object has obviously no access history, and therefore the provider set chosen by the placement algorithm might change in a near future, when the data object has some accesses. Therefore, *Scalia* uses the statistics collected for the class of the object to determine the statistically best set of providers for this new object. Intuitively, a large archive file is most probably a backup, which will not be read often. On the other hand, a small image (such as a logo) will have plenty of read operations. The optimal set of providers for the aforementioned two examples will be different. Thus, thanks to the statistics collected for each class of objects, the probability that the first placement is already optimal increases. As depicted in Figure 6.4, given $row\_key = C(obj)$, the placement algorithm has access to the most probable values regarding the resources that the new object obj will use and its lifetime, and therefore is able to make the best possible placement at this early point.

Let $P(obj) = \{p_i\}$ be the set of storage providers (both public and private) available for storing the data object obj, with $|P|$ being the total number of

Algorithm 6.1 Compute the best provider set for storing the chunks of a data object *obj* based on its access history $H(obj)$.

```
$price \leftarrow MAX\_DOUBLE$
$providers \leftarrow \{\}$
$threshold \leftarrow 0$
$combs \leftarrow \{\}\{\}$
/* combs is a list of provider sets */
$combs \leftarrow getAllCombinations(P(obj))$
for all $pset \in combs$ do
    /* check lockin criteria */
    $lockin \leftarrow 1/|pset|$
    continue if $lockin > obj[lockin]$
    /* check durability criteria */
    $th \leftarrow getThreshold(pset, obj[durability])$
    continue if $th \leq 0$
    /* check availability criteria */
    $av \leftarrow getAvailability(pset, th);$
    continue if $av < obj[availability]$
    $pr \leftarrow computePrice(pset, H(obj))$
    if $pr < price$ then
        $price \leftarrow pr$
        $providers \leftarrow pset$
        $threshold \leftarrow th$
    end if
end for
return $\{providers, threshold\}$
```

providers. A data object has to satisfy several properties contained in the service level agreement (SLA) with the user, such as the minimum durability $obj[durability]$, the minimum availability $obj[availability]$ and the lock-in ratio $obj[lockin]$. Note that the algorithm is not restricted only to these user requirements.

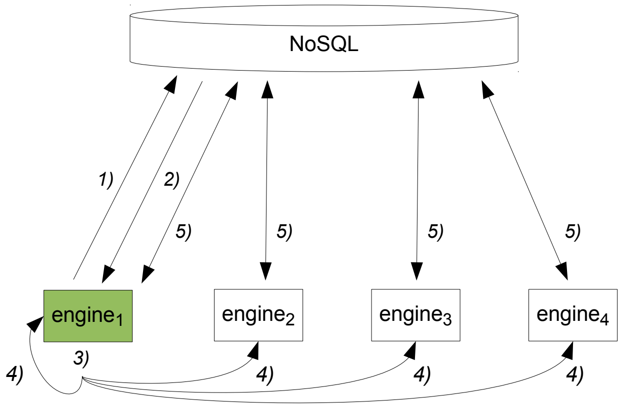
Algorithm 6.1 describes how to compute the best provider set for storing the chunks of a data object *obj* based on its access history $H(obj)$. The function $getAllCombinations()$ returns the list of every combination of the $|P|$ providers available for an object. As described in Algorithm 6.2, the largest value of *m*, as defined in Subsection 6.2.1, for a set of providers is given by $getThreshold()$, so as to satisfy the durability constraint of the object. Let us recall that having a value as large as possible for *m*, referred to as *threshold*, reduces the vendors lock-in and minimizes the storage overhead introduced by the erasure coding of the object. In Algorithm 6.2, starting from zero, the number of failed providers is increased until the durability constraint $obj[durability]$ (*dr* in Algorithm 6.2) is no more satisfied by comparing *dr* with the probability that the object *obj* can be reconstructed from the non-failed providers according to the durability SLA of each provider. When the threshold is equal or less than zero, the set of providers is not able to satisfy the durability constraint. The function $getAvailability()$ computes the availability of the object offered by the set of providers passed in parameter

Algorithm 6.2 *getThreshold()* function: compute the largest threshold given the set of providers *pset* and the required durability *dr*.

```
Require: $pset, dr$
$dura \leftarrow 0$
$failuresOK \leftarrow -1$
$combs \leftarrow \{\}\{\}$
while $dura < dr$ && $failuresOK < |pset|$ do
  $failuresOK \leftarrow failuresOK + 1$
  $upP \leftarrow 0$
  /* getCombinations() returns all combinations (without duplicates)
  of size $'failuresOK'$ from the elements of $'pset$ '*/
  $combs \leftarrow getCombinations(pset, failuresOK)$
  for $comb \in combs$ do
    $upPComb \leftarrow 1$
    for all $p \in pset$ do
      if $p \in comb$ then
        /* provider $p$ has failed */
        $upPComb \leftarrow upPComb * (1 - p[durability])$
      else
        /* provider $p$ has not failed */
        $upPComb \leftarrow upPComb * p[durability]$
      end if
    end for
    $upP \leftarrow upP + upPComb$
  end for
  $dura \leftarrow dura + upP$
end while
return $|pset| - failuresOK$
```

according to their SLA, in order to be compared with the minimum availability requirement $obj[availability]$ of the object. The availability value av is obtained by computing the probability of the object to be successfully reassembled when up to th providers are unreachable. Finally, given the access history of an object, the function $computePrice()$ returns the cost a user has to pay if the object is stored on the provider set given in parameter.

The complexity of Algorithm 6.1 is $O(2^{|P|})$, where $|P|$ is the number of cloud storage providers. However, only the minimally feasible solutions have to be explored, which are much fewer than $2^{|P|} - 1$. As there are currently only a few (less than 15) cloud storage providers available on the market, the optimal solution is still computationally feasible. If the number of providers increases, then suboptimal solutions have to be considered. Actually, this optimization problem resembles the multi-dimensional knapsack problem [96], which is NP-complete. In the knapsack problem, one has to maximize the value of items in a knapsack, while respecting a maximum weight constraint. In our case, we want to minimize price, while satisfying the minimum availability, durability, and lock-in constraints. For any fixed number of constraints, the knapsack problem does admit a pseudo-polynomial time algorithm (similar to the one for basic knapsack) and a polynomial-time approxi-



(engine$_1$ is the current leader)
1) get modified/accessed objects since last optimization
2) A={object key$_1$,object key$_2$,object key$_3$,...}
3) A → {a$_1$, a$_2$, a$_3$, a$_4$}
4) send a$_i$ to engine$_i$
5) for each key in a$_i$, get the statistics and recompute placement only if necessary

Figure 6.5: Periodic Optimization

mation scheme. Optimizing Algorithm 6.1 is left for future work.

Periodic Optimization

Recomputing the placement of every data item may become costly as the number of unique data objects can be very large (e.g., Amazon S3 is reported to store more than 339 billion objects as of June 2011). Iterating over all entries (i.e., a full table scan) is obviously not a scalable solution. Note that the provider set of an object will change only if its access history varies significantly or if the set of storage providers $P(obj)$ changes. Therefore, detecting the changes of the access history pattern of the objects and only optimizing the placement of the objects that may have a new economically-efficient provider set greatly reduces the amount of work and resources needed to continuously ensure that every object is optimally placed. It also permits to run the optimization procedure often, so that the system reacts fast. Moreover, the operational and computational complexity of the placement optimizations should be kept as low as possible in order the solution to remain scalable when the number of managed objects increases.

Periodically (e.g., every 5 minutes), *Scalia* starts the optimization procedure as depicted in Figure 6.5. At time t, a new optimization procedure o_t starts: a leader, elected among all engines from all datacenters, retrieves from the statistics database the set $A = \{obj_i\}$ of object keys that have been accessed or modified after the last optimization procedure o_{t-1}. The leader splits A into $|E|$ subsets of equal size, where $E = \{e_i\}$ is the set of all engines from all datacenters. A subset a_i of keys is assigned to each engine $e_i \in E$. For every object key in a_i, an engine e_i will determine whether the access history pattern of the object has changed or not, by using the *detect*() function described in the Algorithm 6.4. In order to detect a changing access pattern at time t, a statistics window of size 3 [2] is employed, more specifically s_t, s_{t-1} and s_{t-2}. The algorithm also takes as input a threshold *limit*, the decision period d of the object as well as several metadata computed during previous optimizations, such as the timestamp of the last placement recalculation or the moving average values when the last trend change of a statistic was detected.

Algorithm 6.3 Trend detection: *alert()* function

```
Require: $v1, v2, limit$
res = 0.0
if $(v1 == 0 || v2 == 0)$ then
    return $false$
end if
if $(v1 > v2)$ then
    $res = \frac{v1}{v2}$
else
    $res = \frac{v2}{v1}$
end if
if $(res > limit)$ then
    return $true$
else
    return $false$
end if
```

Only if the access history pattern has changed considerably (based on *limit*), the engine will recompute the placement of the object using Algorithm 6.1. If a better provider set is found and if the cost of migration is covered by the benefits of migrating to the new provider, it will migrate the chunks accordingly. The placement of objects with no access or a non-varying access pattern will not be recomputed. Figure 6.6 and 6.7 show when the object placement is recomputed, given a real website access pattern (the website has around 2500 visitors per day mainly coming from Europe (62%), North America (27%) and Asia (6%)).

As an engine itself is completely stateless and independent, adding more computing power is straightforward. Moreover, in order not to deteriorate the reactivity and the performance for handling the clients requests, the code performing the optimization process can easily be realized as a standalone service and can run on distinct servers.

6.3.2 Caching Layer

In order to improve the reactivity of the read operations, *Scalia* maintains a distributed (per datacenter) cache layer. Upon a data read, if the data is

[2] The window should be as small as possible to limit the computational complexity, while still being long enough to detect a changing pattern.

Algorithm 6.4 *detect()* function: detect if the access trend has changed, and allows the recomputation of the object placement.

```
Require: $s_t, s_{t-1}, s_{t-2}$ /* access stats */
Require: $limit$ /* threshold limit */
Require: $D_{obj}$ /* decision period */
Require: $meta$ /* array with the object metadata:
    - last moving avg (MA) for each resource: bdwin, bdwout, storage, ops
    - timestamp of last placement (PL) recalculation */
resources ← {bdwin, bdwout, storage, ops}
for all $r \in resources$ do
    $curMA \leftarrow \frac{s_t[r]+s_{t-1}[r]+s_{t-2}[r]}{3}$
    if $alert(curMA, meta[MA][r], limit)$ then
        $meta[MA][r] \leftarrow curMA$
        if $now() - meta[PL][r] > D_{obj}$ then
            $meta[PL][r] \leftarrow now()$
            return {true, meta}
        end if
    end if
end for
return {false, meta}
```

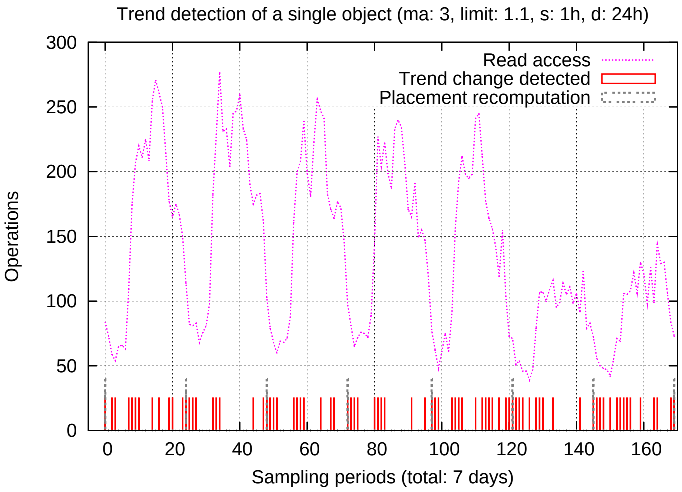


Figure 6.6: Trend detection using a threshold limit of 1.1, a sampling period of 1 hour, a decision period of 1 day (24 hours) and an history moving average of 3. The access statistics come from a real website.

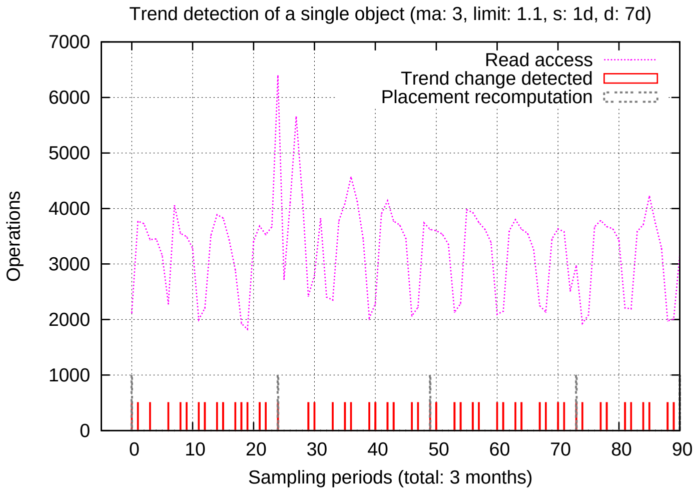


Figure 6.7: Trend detection using a threshold limit of 1.1, a sampling period of 1 day, a decision period of 1 week (7 days) and an history moving average of 3. The access statistics come from a real website.

present in the cache, there is no need to fetch the chunks from the remote providers and reassemble the data object before serving it to the client. Otherwise, the data is reassembled from the chunks, served to the client and stored in the cache. Not only this layer reduces the requests latency, but it also reduces the interactions with the storage providers, resulting in lower costs for the user. In a multi-datacenter setup, the cache has to be invalidated in all datacenters in order to guarantee the consistency of the read operations. The caching layer can be combined and extended by a CDN to reach even better read performance.

6.3.3 Database Layer

The database layer of *Scalia* stores the metadata (i.e., the rules set by the end users regarding the durability, availability or vendor lock-in avoidance constraints of their data objects, the public provider settings, the settings of the users' private storage resources) as well as the access history of the data objects (i.e., the statistics). The database can be concurrently accessed by several engines updating the same entry, in all datacenters. As clients' requests are routed to all datacenters indifferently, the database has to be replicated; the classic master-slave replication scheme of traditional databases is not suitable for our multi-master setup, as *Scalia* has to keep working even when a datacenter is down. Moreover, not only the read but also the write operations have to be scalable. Therefore, we consider here a NoSQL database [4], which have a better support for multi-datacenter deployment and network/server failures.

Concurrency and Conflicts

In a distributed system, a race condition can result in catastrophic situations where concurrent updates for the same entry can lead to data corruption or data loss. To deal with concurrency, two approaches are imaginable in our architecture. The first solution is to use a distributed locking mechanism, such as Zookeeper [107], to ensure that an entry is updated only by a single engine at a time. However, because of our multi-datacenter setup, Zookeeper needs to be synchronized among the datacenters and results in higher write operation latency. Even worse, in case of a network partition between the datacenters, Zookeeper is not able to form a quorum and assign locks. To solve this issue, a third party, monitoring all datacenters and assigning the role of a master to one datacenter is required in case of failure. The detailed setup of this architecture is outside the scope of the chapter, and will not be discussed here.

Multi version concurrency control (MVCC) is an alternative approach without locks, where an update operation does not delete the old data overwriting it with the new one. Instead, the old data is marked as obsolete and the new version is added, resulting in storing multiple versions of the data with only one being the latest. If an entry is updated concurrently in multiple datacenters, the database will detect the conflict (e.g., employing anti-entropy mechanisms such as vector clocks). The user will be prompted to decide which version is the good one and *Scalia* will remove the other version. Alternatively, *Scalia* can decide by itself to keep only the latest version without asking the end user, however it requires that each engine is time-synchronized.

Statistics

The read and write accesses of an object are collected using a distributed and reliable service (e.g., Flume [8] or Scribe [14]) for efficiently collecting, aggregating, and moving large amounts of log data: a *log agent* residing at each engine continuously reads the logs containing the statistics of the requests handled by the engine, and sends them to one of the *log aggregators*. The latter collect and aggregate the logs before writing them to the database.

The placement algorithm also needs statistics about the objects managed by *Scalia* to take pertinent decisions when there is no access history of new objects, or when it has to predict the deletion of an object in order to optimize its placement. Those statistics are obtained using map-reduce jobs on the database, so as to aggregate the statistics of each individual objects.

6.3.4 Life cycle of read and write operations

Scalia relies on multi version concurrency control (MVCC) to deal with concurrent updates and requires every engine to be time-synchronized (e.g., using NTP [35]) in order to resolve conflicts.

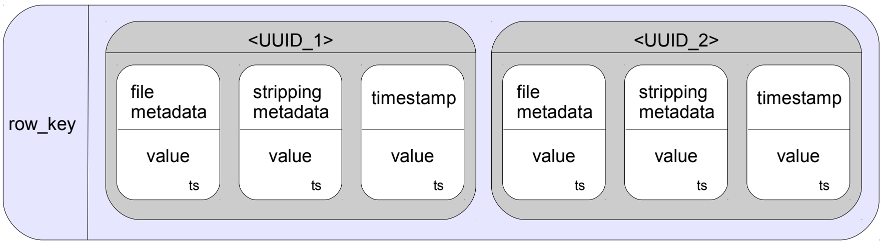


Figure 6.8: Concurrent writes: the *row_key* entry has been updated concurrently, resulting in 2 versions of its metadata. When the conflict is detected, the chunks corresponding to the oldest version are removed from the storage providers, and the oldest version is removed from the database.

Write Operation

During a write operation of an object *obj*, a user will provide at least the following input through the *Scalia* interface: a container name $obj[container]$, a key $obj[key]$ and the data $obj[data]$. After having decided the optimal set of providers $P(obj)$, *Scalia* splits the data object into $|P(obj)|$ chunks, and stores the latter at the selected storage providers using as key:

$$skey = MD5(obj[container] \mid obj[key] \mid UUID)$$

$UUID$ is a globally unique identifier which prevents concurrent updates to cause data corruption. The metadata of *obj* is written to the database with $UUID$ as the primary key, as depicted on Figure 6.8. As row key for writing the metadata, *Scalia* uses:

$$row\_key = MD5(obj[container] \mid obj[key])$$

Table 6.5 shows an example of metadata stored for an object.

If the write operation is an update, older metadata corresponding to *obj* is discarded and the corresponding chunks deleted from the providers. The operations are logged and will be processed by the distributed log system, in order to be written in the statistics database. When a conflict is detected by the database in case of concurrent writes, the timestamps are compared, and only the freshest version is kept; the deprecated version of the object is removed from the storage providers and from the statistics database. Note that writing the statistics never conduct to conflicts in the database thanks to an adapted data model, where statistics are always written using globally unique keys.

Read Operation

To read an object *obj*, the end user sends a request to the *Scalia* API with the container name and the object key as parameters. The randomly chosen engine that has received the request checks first if the data is in the cache. If so, then the data is directly returned from the cache. Otherwise, the engine reads

Table 6.5: Metadata of the file `myvacation.gif`

Striping metadata	*File metadata*
chunk1: provider_2	name: myvacation.gif
chunk2: provider_5	mime: image/gif
chunk3: provider_7	checksum: ce944a11a4
chunk4: provider_1	size: 342 KB
m: 3	policy: rule_3
skey: a3e229084	container: pictures

the metadata of obj from the database, retrieves the m out of $|P(obj)|$ chunks from the cheapest (other criteria can be considered) providers, reassembles the data and sends it to the client. The data is also stored in the cache. The operations are logged and stored in the statistics database.

Error Handling

At the providers' side It may happen that one of the storage providers is not available. If it happens during a write operations, *Scalia* will choose the best placement that does not include the faulty provider. In case of a read operation, if $|P(obj)| > m$, then the data can still be retrieved from the m storage providers available. Recall that m corresponds to the minimum amount of chunks needed to reconstruct a data item. Finally, for a delete operation, the deletion of the chunk residing at a faulty provider is postponed until the provider recovers. As we employ the MVCC approach, incomplete operations do not introduce inconsistencies.

At Scalia side Within a single datacenter, no layer has a single point of failure. In a multi-datacenter setup, where requests are routed indifferently to each datacenter, the database layer might cause a problem. In fact, thanks to an advanced support of multiple datacenters, the NoSQL database automatically stores a replica in multiple datacenters. Therefore, read requests sent to the *Scalia* API can always be served. Regarding write requests, as long as a single database node is up and running, no operation will fail, and when the second datacenter recovers, the replicas in the various datacenters will be eventually consistent.

6.3.5 Private Storage Resources

An interesting property of *Scalia* is the ability to use private storage resources together with commercially available public cloud storage solutions. Corporate storage resources (workstations, servers, NAS, SAN, ...) or dedicated servers can be registered to *Scalia* with a description of their properties: amount and price of available storage, price of incoming and outgoing bandwidth and price per operation. The placement algorithm will take into account these new resources to minimize the costs of storing and serving the user's data. Thanks to *Scalia* and its unified interface, it is straightforward to use local resources up to their capacities, and then use the best suited provider(s) when demand grows.

In order for a private storage resource to be accessible from *Scalia*, a standalone web service needs to be deployed locally on the resource. The web service is a lightweight and standalone web server that offers an authenticated Amazon S3 compatible REST interface to store and retrieve files. The data is stored on the local filesystem or on any distributed/parallel filesystem (NFS [140], MogileFS [29], . . .) accessible directly from the web service and will never grow beyond the limit set in the properties of the resource. A private token generated by the private resource owner is also registered to *Scalia*, so that only legitimate requests are considered by the web service. The authentication is done by signing the request (i.e., HMAC of the requests parameters using the private token) and to prevent replay attacks, a timestamp is also included in the request. If the data stored is sensitive, the web service can be configured to use SSL/TLS.

6.3.6 Discussion

A primary objective of *Scalia* is to reduce the storage costs for end users. However, running *Scalia* as local appliance or using a (hosted) brokerage service involves additional costs. The benefit from using *Scalia* should cover the customer's extra operational expenses. This is highly expected as the complexity of our simple-by-design algorithms does not depend on the amount of managed data objects, but only on the window size and on the finite number of cloud storage providers. To this end, the *Scalia* service fee (or cost) could be seen as a percentage of the storage costs saved by *Scalia* on behalf of the customers, and thanks to the economies of scale (i.e., same *Scalia* resources can serve a large number of customers), *Scalia* should be a sustainable solution. Describing a detailed business model is outside the scope of the chapter.

A second important objective of *Scalia* is to avoid vendor lock-in. At first sight, it may seem that customers become locked in a specific *Scalia* broker. However, as *Scalia* exposes, via an authenticated API, the metadata of the managed objects (e.g., the chunk locations for a given object), a customer can completely bypass *Scalia* and access its data directly from the cloud storage providers. To this end, stopping to use *Scalia* or using another *Scalia* broker does not involve any switching costs.

One may argue that using *Scalia* as a brokerage service may add additional network latency. In fact, *Scalia* could even decrease the latency for retrieving objects, as the data chunks are fetched in parallel from the fastest providers (i.e., from m out of n), instead of sequentially downloading data from a single and potentially slow one. In order to further improve the read performance, *Scalia* can be extended to use a CDN to deliver the content more efficiently and with improved availability of the objects due to caching at the edge servers of the CDN. Moreover, as most storage providers in the market combine their storage solution with a CDN one, *Scalia* could also provide the cheapest storage provider / CDN provider pair, based on the data access patterns and the customer's storage/access latency constraints.

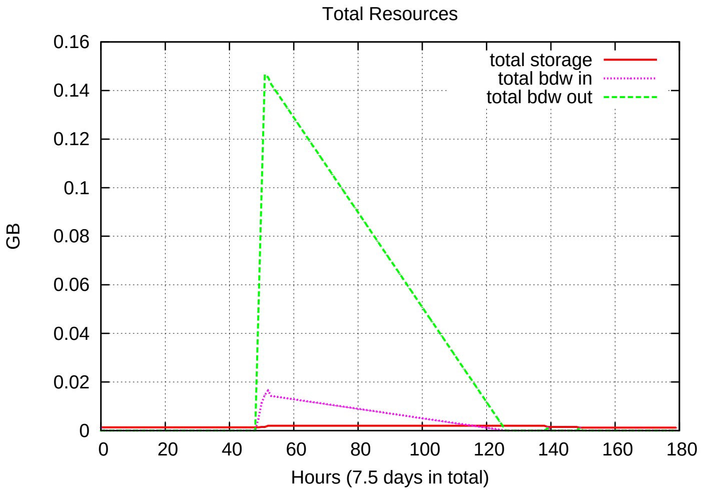


Figure 6.9: Slashdot scenario: total amount of resources from the storage providers used by *Scalia* to store and serve the object

6.4 Evaluation

6.4.1 Experimental Setup

Table 6.6: Sets of providers (c.f Table 6.3 for abbreviations)

#	*Set of Providers*	#	*Set of Providers*
1	S3(h)-S3(l)	14	S3(h)-Ggl-RS
2	S3(h)-S3(l)-Azu	15	S3(h)-RS
3	S3(h)-S3(l)-Azu-Ggl	16	S3(l)-Azu
4	S3(h)-S3(l)-Azu-Ggl-RS	17	S3(l)-Azu-Ggl
5	S3(h)-S3(l)-Azu-RS	18	S3(l)-Azu-Ggl-RS
6	S3(h)-S3(l)-Ggl	19	S3(l)-Azu-RS
7	S3(h)-S3(l)-Ggl-RS	20	S3(l)-Ggl
8	S3(h)-S3(l)-RS	21	S3(l)-Ggl-RS
9	S3(h)-Azu	22	S3(l)-RS
10	S3(h)-Azu-Ggl	23	Azu-Ggl
11	S3(h)-Azu-Ggl-RS	24	Azu-Ggl-RS
12	S3(h)-Azu-RS	25	Azu-RS
13	S3(h)-Ggl	26	Ggl-RS
		27	*Scalia*

As we mainly discuss the costs involved in several setups, we only present here results coming from a simulator. The availability and durability guarantees of the five public storage providers considered in this evaluation are described in Table 6.2. Their pricing policies regarding the costs of resources such as storage, incoming and outgoing bandwidth are described in Table 6.4. In the following experiments, we consider without loss of generality that Amazon S3 with a specific high durability (i.e., S3(h)) is completely indepen-

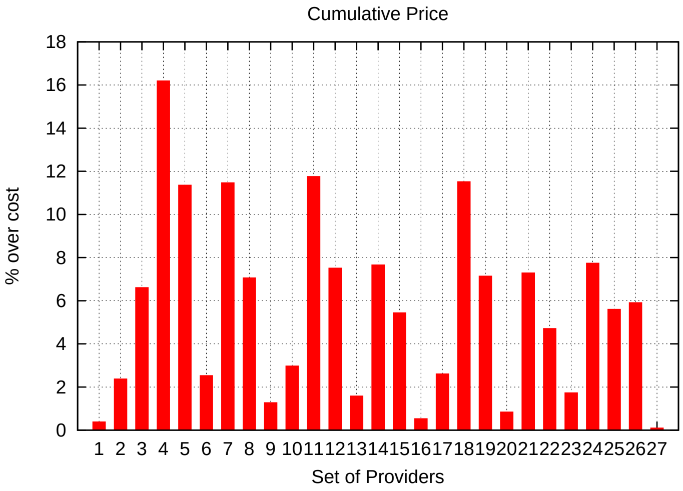


Figure 6.10: Slashdot scenario: total cost after a week of the provider sets that satisfy the constraints of the object. The labels of the provider sets are given in Table 6.6. *Scalia* is number 27.

dent from Amazon S3 with a specific low durability (i.e., S3(l)), and there are no correlated failures or any relation between them.

We compare the cost of multiple static sets of providers with the cost of the dynamic set of providers chosen by *Scalia*. As a baseline, for every sampling period, we compute the optimal placement, which corresponds to the cheapest set of providers in term of resources consumed (storage, number of operations, incoming bandwidth, outgoing bandwidth) for handling the load during that period. Given the optimal set of providers for a period, we then compute the corresponding optimal cost and the percentage of over cost of the different providers' sets.

6.4.2 Slashdot Effect Scenario

In this experiment, we simulate the behavior of the "Slashdot effect", where suddenly an object becomes highly popular and starts to receive a lot of requests. After 2 days (48 hours), the number of read requests goes from 0 to 150 in only 3 hours, and then slowly decreases at the rate of 2 requests per hour. The object stored has size 1MB, a minimum availability of 99.99% and durability of 99.999%. The durability constraint is easily met by only 1 provider; however, the availability constraint requires at least 2 providers. As depicted in Figure 6.10, *Scalia* is only 0.12% more expensive than the optimal placement. This difference is explained by the cost of the migration of several chunks. *Scalia* uses [S3(h), S3(l), Azure, RS; m:3] before the Slashdot effect. During the requests peak, the cheapest provider set is [S3(h), S3(l); m:1]. When the flash crowd effect is over, *Scalia* chooses [S3(h), S3(l), Azure, Google, RS; m:4] as its provider set. The best static provider set is a mix of [S3(h), S3(l); m:1] which is 0.4% more expensive, while the worst static

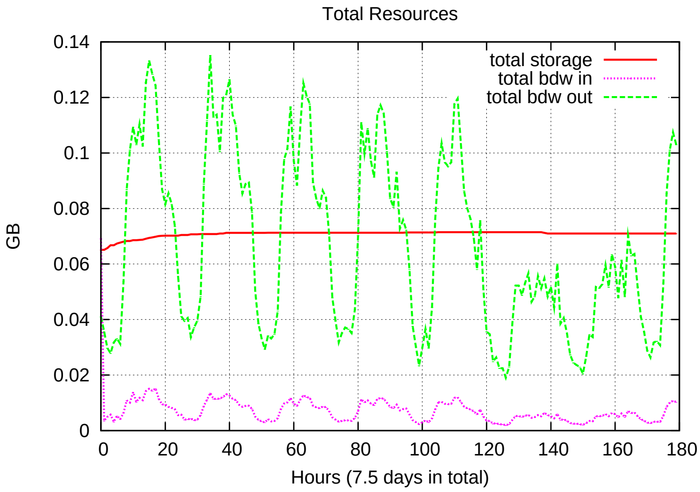


Figure 6.11: Gallery scenario: total amount of resources from the storage providers used by *Scalia* to store and serve the pictures

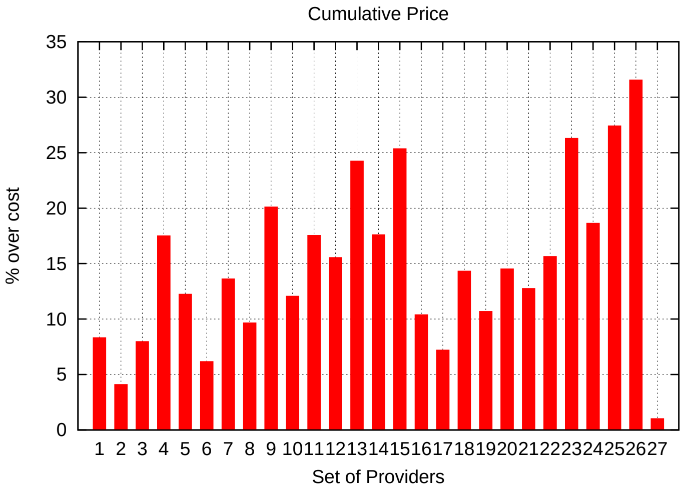


Figure 6.12: Gallery scenario: total cost after a week of the provider sets that satisfy the constraints of the object

provider set [S3(h), S3(l), Azure, Google, RS; m:4] is 16% more expensive than the optimal placement.

Thanks to the adaptivity of *Scalia*, the best provider set for a given access pattern is always chosen.

6.4.3 Gallery Scenario

In this scenario, 200 pictures (250 KB each) have to be stored. The pictures are accessed following the daily pattern of a real website which has around 2500 visitors per day mainly coming from Europe (62%), North America (27%) and Asia (6%). Moreover, the popularity of the pictures follows a Pareto (1,50) distribution. The minimum availability per picture is set to 99.99%.

Ideally, all pictures should not be stored to the same set of providers, because some pictures are popular and the cost of storage is negligible as compared to the cost of outgoing bandwidth. On the other hand, unpopular pictures should be stored to the provider set with the lowest storage cost, while still ensuring the availability and durability constraints.

Figure 6.11 depicts the total amount of resources used by *Scalia* for storing and serving the pictures from the different storage providers. In Figure 6.12, *Scalia* is only 1.06% more expensive than the optimal placement and outperforms all the other static sets of providers. The best static set of providers is 4.14% more expensive, while the worst set is 31.58% more expensive than the optimal placement.

The popular pictures mainly use [S3(h), S3(l); m:1], moderately popular pictures use [S3(h), S3(l), Azure; m:2] and unpopular pictures use [S3(h), S3(l), Azure, Google; m:3]. Therefore, it clearly appears that storing all pictures to the same set of providers results in over-charging. The adaptivity of *Scalia* dynamically finds the most cost-efficient placement of an object based on its access pattern. Therefore, an end user does not need to decide a fixed placement per data object by guessing the access pattern of the object.

6.4.4 Adding Storage Resources

Public Storage Resources

We now consider a scenario where a new object of 40 MB needs to be stored every 5 hours. Unlike preceding scenarios where the availability constraint was important, here the data owner wants to avoid vendor lock-in and therefore each object has to be stored at 2 different providers at least.

At hour 400 a new storage provider *CheapStor* is registered in the system and offers an attractive storage alternative: 0.09$ per GB of storage, 0.1$ per GB of bandwidth in, 0.15$ per GB of bandwidth out and 0.01$ for 1K of operations.

Before hour 400, *Scalia* stores the objects using [S3(h), S3(l), Azure, Google, Rackspace; m:4]. After the new provider has been registered, *Scalia* migrates the already stored objects and stores the new objects to [S3(h), S3(l), Azure, CheapStor, Rackspace; m:4]. The total amount of resources used in this experiment is shown in Figure 6.13.

In this scenario, *Scalia* dynamically adapts to the changing conditions (a new provider has shown up) and is only 0.35% more expensive than the optimal solution, as depicted in Figure 6.14. The best static placement [S3(h), S3(l), Azure, Google, Rackspace; m:4] is not able to take into account the new provider, and therefore costs 7.88% more than the optimal placement. Finally, the worst static placement is 96.35% more expensive! Figure 6.14 shows clearly that having a large number of providers is really an advantage when using erasure coding for the storage, as the storage overhead factor $\frac{1}{r} = \frac{n}{m}$ will tend to 1.

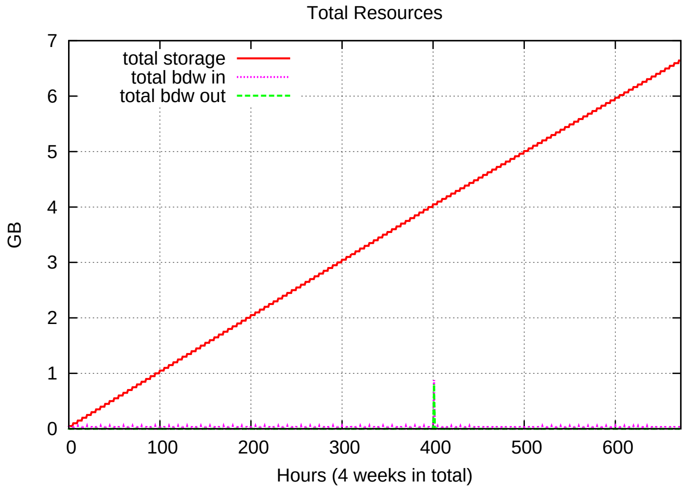


Figure 6.13: Adding a public storage provider: total amount of resources used by *Scalia* to store the backup objects to the storage providers

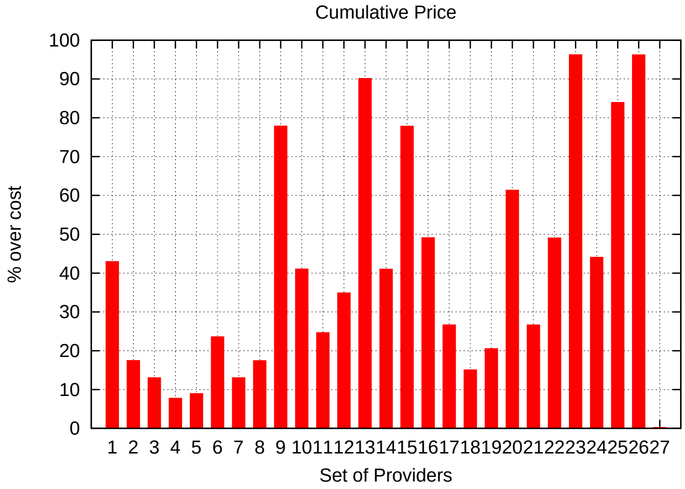


Figure 6.14: Adding a public storage provider: total cost after a month of the provider sets that satisfy the constraints of the object

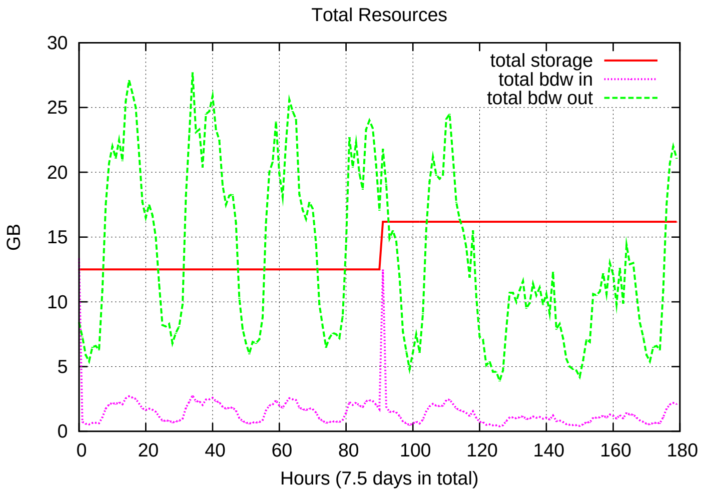


Figure 6.15: Adding a private storage resource: total amount of resources used by *Scalia* to store the objects to the storage providers

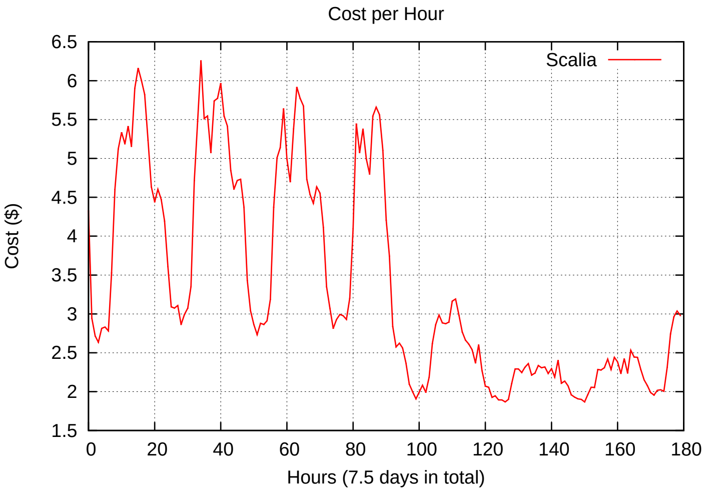


Figure 6.16: Adding a private storage resource: total cost per hour with *Scalia*. After hour 90, *Scalia* uses the private storage decreasing the total cost.

Private Storage Resources

In this scenario, we store 100 objects of $\frac{1}{10}$ GB. At hour 90, the data owner registers to *Scalia* a private storage resource of 5 GB only with storage/bandwidth-in/bandwidth-out costs of 0.05$/GB. The objects are accessed following a daily pattern using a Pareto(1,50)-based popularity.

As soon as the new provider is registered, *Scalia* moves all objects that fit in the available private storage, and leaves the remaining objects where they are currently stored. Figure 6.15 shows the total amount of resources used

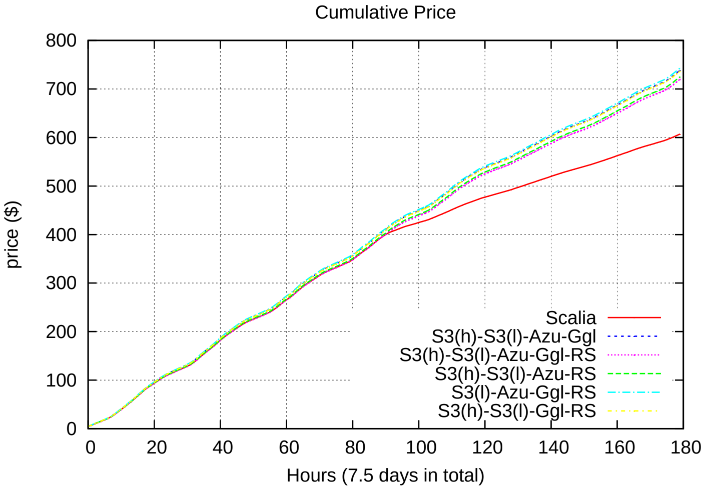


Figure 6.17: Adding a private storage resource: cumulative cost over a week of the cheapest providers sets (whose total cost $<$ 123% of optimal cost) that satisfy the constraints of the object.

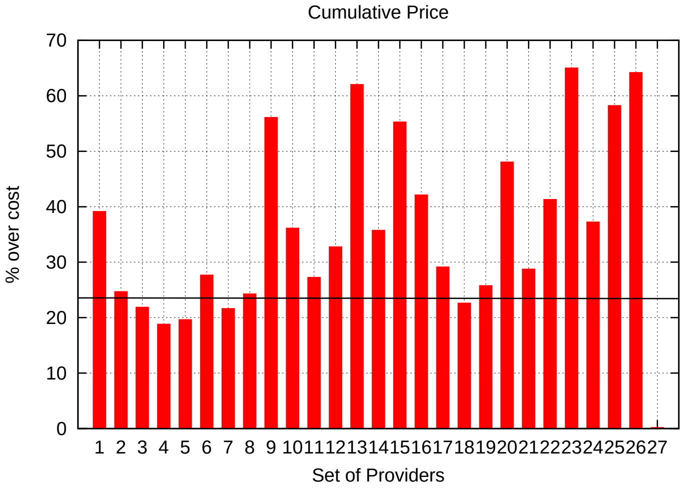


Figure 6.18: Adding a private storage resource: total cost after a week of the providers sets that satisfy the constraints of the object. The horizontal line is at 23% over cost.

by *Scalia*. At hour 90, up to 5 GB of objects are moved to the private storage resource, and thus a peak of incoming bandwidth can be observed. Moreover, the total amount of storage increases. Before hour 90, all objects were stored using [S3(h), S3(l), Azure, Google, Rackspace; m:4]. The total storage space needed to store all objects is

$$100 * \frac{5}{4} * obj[size].$$

After hour 90, 47% of the keys use [S3(l), Private; m:1], 6% use [S3(h), S3(l), Private; m:2], and the remaining 47% still use the initial providers set. Because more than half of the objects are now split into fewer chunks, the extra storage needed to store the objects is now around 29% greater:

$$100 * (\frac{47}{100} * \frac{5}{4} + \frac{47}{100} * \frac{2}{1} + \frac{6}{100} * \frac{3}{2}) * obj[size]$$

Figure 6.16 shows the cost per hour using *Scalia*: after hour 90, the cost per hour decreased thanks to the lower price of the private resources. Figure 6.17 shows the cumulative price over time of *Scalia* compared to the five cheapest sets of providers, which have an over cost less than 23% of the optimal price. Finally, Figure 6.18 depicts the total cost of every set of providers after the experiment.

6.4.5 Active repair

In the case of a transient failure of a cloud storage provider, *Scalia* may adopt two strategies to cope with the unavailability of providers: either do nothing and simply wait for the provider to recover, or move the chunks hosted at the faulty provider to another provider. However, the latter procedure comes at a relatively high cost: in order to move the chunk of the faulty provider, the data object needs to be reconstructed from the remaining chunks and split again into chunks. Depending on the available providers, the threshold m of the most cost-effective providers set may be different. In that case, all chunks need to be re-written. If m is the same, then only the faulty chunk needs to be written, which corresponds to the cheapest case.

Like in Section 6.4.4, we consider a scenario where a new object of 40 MB needs to be stored every 5 hours. At hour 60, one of the provider, S3(l), has a transient failure and is not reachable anymore. At hour 120, the provider is again up and running.

Scalia is compared to the static provider set [S3(h), S3(l), Azu; m:2]. Before hour 60 and after hour 120, *Scalia* uses [S3(h), S3(l), Azu; m:2] as well. During the unavailability of S3(l), *Scalia* uses another provider to store the unreachable chunk: [S3(h), Ggl, Azu; m:2]. However, in the static provider set, the unreachable chunk cannot be moved to another provider, and therefore the data needs to be split into only 2 chunks, resulting in using [S3(h), Azu; m:1] during the failure period. Figure 6.19 shows the cost difference of active repair between the fixed and the dynamic sets of providers.

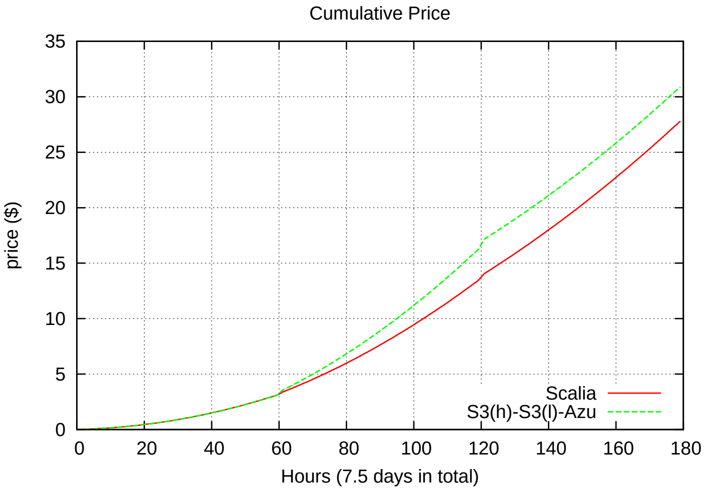


Figure 6.19: *Scalia* versus a fixed set of providers during active repair.

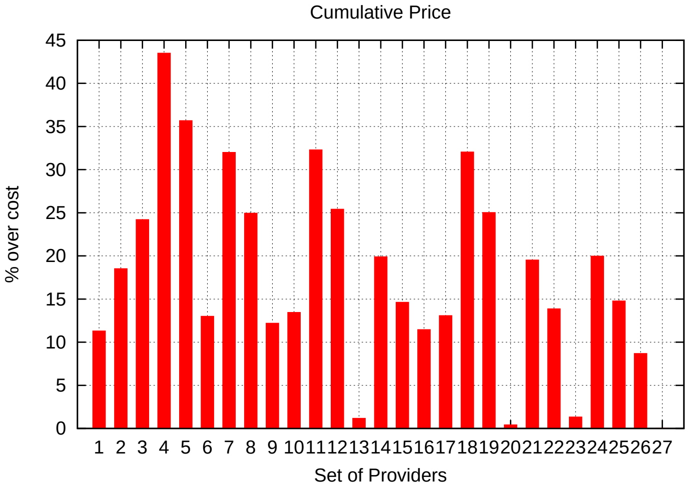


Figure 6.20: At hour 60, the provider *Ggl* decreases its pricing. *RS* reacts at hour 120 by also decreasing the price of its offer.

6.4.6 Pricing Update

When a provider changes its pricing, the best providers set for a data may also change. *Scalia* will cope with it and will automatically update the placement of data that will gain from the new pricing policy. In this scenario, the provider *Ggl* adjusts its tariffs at hour 60, in order to be one of the cheapest providers: the storage price decreases from 0.17 USD/GB to 0.1 USD/GB, the bandwidth in price goes from 0.1 USD/GB to 0.08 USD/GB, the bandwidth out price goes from 0.15 USD/GB to 0.12 USD/GB. At hour 120, the provider *RS* reacts to the price change of *Ggl* by also reducing its owns pricing: the storage price is now 0.09 USD/GB, the bandwidth-in is 0.07 USD/GB and the bandwidth-out 0.11 USD/GB.

In this scenario, 100 files (500 KB each) have to be stored. The files are accessed following the daily pattern of a real website which has around 2500 visitors per day mainly coming from Europe (62%), North America (27%) and Asia (6%). Moreover, the popularity of the files follows a Pareto (1,50) distribution. The minimum availability per file is set to 99.99%.

Again, as depicted in Figure 6.20, *Scalia* adapts to the changing situation and, based on the access pattern of the data, it chooses the best provider set. Here, *Scalia* is only 0.06% more expensive than the dynamic ideally optimal placement. This nice result is obtained, because the data was already optimally placed before the pricing update of the providers.

6.5 Related Work

Scalia was inspired by RACS [57], which employs RAID at the cloud storage level, making also use of erasure codes [82] instead of full replication [159]. HAIL [67] distributes redundant blocks of a file across multiple servers, while allowing a client to make sure that the file is not corrupted even in the case of a server compromise. RAID-like techniques have already been used by several P2P storage systems [72, 74] to ensure durability and availability of data. Storage providers like [52] use an hybrid model where data is split into redundant blocks at client side (via erasure coding); the blocks are distributed to other end users following a P2P approach and also sent to the servers managed by the provider. This approach increases durability of the data, while decreasing the storage costs of the provider. Other commercial providers like [6] also make use of erasure coding to disperse data blocks to several geographical distinct regions over the world, providing very high durability guarantees. All the aforementioned approaches do not improve the placement of the data objects according to their access pattern.

Commercial network appliances or servers [7, 32] residing at the customer, called cloud storage gateway, can also serve as intermediaries to multiple cloud storage providers. While they include storage features such as caching, backup, recovery, encryption or de-duplication, these systems do not take into account the access pattern of the data nor its expected lifetime to continuously choose the optimal providers set.

Also, several libraries [23, 24] for accessing public cloud storage and cloud computing infrastructures with a unified interface are quickly emerging, thus showing the increased need of avoiding vendor lock-in.

Finally, extensive previous work [56, 69] is available in the area of job scheduling in computational grids, so as to minimize the cost for the end-users, while satisfying the performance constraints. However, most of these works depend on prior knowledge of the detailed computational cost of a new job and the job placement is fixed. Several approaches on modeling job arrivals [118, 124] and predicting cost amortization [109] using statistics on query traffic have also been proposed. In *Scalia*, data placement is adaptive to the various pricing and resource conditions, so as to dynamically find the optimal data placement.

6.6 Conclusions

We have presented *Scalia*, a system that continuously optimizes the placement of data stored at multiple cloud providers, based on their access statistics. *Scalia* mediates data placement across multiple cloud providers and helps the data owners to avoid vendor lock-in and satisfy certain availability and durability constraints in a cost-effective way. We described in detail the various layers of our approach and our scalable mechanism for adaptive data placement. By extensive simulation experiments, we proved that our solution finds an optimal (i.e., cheapest) data placement for dynamically changing data access patterns and when different cloud providers or different prices are available. The evaluation of the performance and the scalability of the prototype is left for future work. Finally, we will study the employment of the proposed brokering approach for computational resources.

Part IV

Conclusion

Chapter 7

Conclusion

7.1 Summary of the Work

Regarding the database layer, we have addressed the problem of highly skewed popularity of data items by adapting the allocated resources, while ensuring the geographical diversity of replicas. We described *Skute*, a robust, scalable and highly-available key-value store designed to provide high and differentiated data availability statistical guarantees to multiple applications in a cost-efficient way. It dynamically adapts to varying query load or failures by determining the most cost-efficient locations of replicas of data partitions with respect to their popularity and their client locations. Moreover, it efficiently and fairly utilizes cloud resources by performing load balancing in the cloud adaptively to the query load. We experimentally proved that it converges fast to equilibrium, where as predicted by a game-theoretical model no migrations happen for steady system conditions. Our approach achieves net benefit maximization for application providers and therefore it is highly applicable to real business cases. A fully working prototype has been built, which clearly demonstrates the feasibility, the effectiveness and the low communication overhead of our approach in a distributed setting.

On the application layer's side, we took advantage of the benefits offered by the cloud computing paradigm to dynamically adapt the resources, so as to satisfy performance guarantees. We also have addressed problems inherent to cloud computing environments, such as unreliable and ephemeral resources. We proposed *Scarce*, an economic, lightweight approach for dynamic accommodation of load spikes and failures for composite web services deployed in clouds, so as to satisfy performance and availability guarantees. Our cost-efficient approach for dynamic and geographically-diverse replication of components in a cloud computing infrastructure effectively adapts to load variations and offers service availability guarantees. We derived performance constraints per component and we scale up existing virtual machines or create new ones whenever the constraints are not met. Application components act as individual optimizers and autonomously replicate, migrate across virtual machines or terminate based on their economic fitness. Their resource inter-dependencies are implicitly taken into account by means of server rent prices. The requests are routed across components based on their respective prior performance. Our approach also detects unstable cloud resources and reacts accordingly by removing or replacing them, so as to minimize the end-to-end application response time.

Finally, we have addressed an important problem occurring with cloud storage, namely vendor lock-in, while further improving the scalability of cloud storage by federating storage providers. We have presented *Scalia*, a system that continuously optimizes the placement of data among multiple cloud providers subject to optimization objectives, such as cost minimization or budget keeping. *Scalia* mediates data placement across multiple cloud providers and helps the data owners to avoid vendor lock-in and satisfy certain availability and durability constraints in a cost-effective way. Data placement is adapted according to the real-time data access patterns. We described in detail the various layers of our approach and our horizontal scalable architecture for adaptive data placement. By extensive simulation experiments, we proved that our solution finds an optimal data object placement for dynamically changing data access patterns and when different cloud providers or different prices are available.

To summarize, we have dealt with the robustness and the scalability of the database layer by enhancing a key-value store, we have provided a framework to build autonomic cloud-ready applications and we have built a clever cloud storage broker following the best practises in terms of high-availability and scalability.

7.2 Future Work

7.2.1 Improving the Current Work

Several aspects could be addressed by future work. Regarding *Skute* (see Chapter 4), we will investigate the employment of our approach for more complex data models, such as the one in Bigtable [71]. Second, we will look for specific use cases where our approach could lead to large improvements compared to actual solutions, especially in the field of large-scale caching of static data objects.

In relation to *Scarce* (see Chapter 5) and the application layer, we intend to explore our economic paradigm for autonomic resource management in the context of multiple competitive or cooperative cloud providers. Another possible research direction is to allow the self-tuning of service components with heavy data dependencies. A third interesting research direction is to enable cloud applications to be deployed and executed closer to the end users, by running replicas of components directly on edge servers (i.e., on "caching" servers close to the end users), bringing to applications the same advantages that CDNs provide for static content. As the proximity to end users is critical to performance, moving not only static but also dynamic content (generated by the components) close to the user is essential.

Finally, we plan to evaluate the performance and the scalability of the *Scalia* (see Chapter 6) prototype. We will also study the employment of the proposed brokering approach for computational resources. Moreover, *Scalia* can be extended to take into account the use of a CDN to deliver the content and to provide the cheapest storage provider / CDN provider pair, based on the access pattern of the content and the storage/access latency constraints associated with it.

7.2.2 Future Directions

Challenges of Cloud Computing Our dependence towards cloud computing increases every day. In the future, our entire digital life will be mostly in the hands of third-party service providers. Today, many users already rely on cloud-based service providers to manage their social relations, contacts, emails, pictures, videos and even to backup their entire computer. While most of these providers make use of encryption to secure the data transfer from the user to the cloud, they still have a full access to the data. Preserving our privacy and protecting our data against any improper access is not only a technical challenge which will require a major effort but it is also an organizational concern where every actor should cooperate in order to secure our digital life from end to end. The data produced by a user day after day, such as pictures, videos, articles, comments or browsing history should remain the sole property of the user and not of any service provider. If required by the user, a cloud provider should not be able to alter or to have access to his data.

In addition, data should be moved as easily in the cloud as out the cloud, which is rarely the case today where many service providers do not give a simple way (if any) to get his own data back. Ensuring that data can be migrated from a service provider is all the more necessary that providers can change their terms of use or their privacy policy at any time. A standardization effort is therefore required in order to allow a safe adoption of cloud-based services, where data can be exported from one provider and imported to another one in a straightforward manner.

Similarly, cloud computing providers should offer greater transparency of their services, as it is necessary to effectively monitor each layer of an application to ensure that the price charged by a provider is in line with the delivered performance and capacity.

Opportunities of Cloud Computing A standardized way to interact with service providers opens interesting research areas such as federation of computing and storage resources offered by different providers. Moreover, providers could be switched on the fly according to specific objectives, allowing to create new services by simply combining the services offered by distinct providers, similarly to Web Service composition.

Today, cloud computing seems to offer infinite capacity. However, as the available computing resources are not infinite and as the demand is quickly growing, a global marketplace of computing, storage and bandwidth resources may emerge in the near future. As in other marketplaces, the price of a resource will vary depending on supply and demand. Creating a worldwide computing marketplace undoubtedly presents interesting research opportunities.

Delivering content to computers across the planet is a well addressed problem: CDNs or peer-to-peer networks already provide an efficient way to communicate with a vast number of users via Internet. However, with the rise of mobile computing, new protocols or architectures better adapted to mobile and wireless devices, such as smartphones or tablets, should take into account their specific characteristics, such as wireless connectivity, smaller screen, limited computing, storage and energy resources. While transcoding (of videos for example) is a typical example of cloud computing usage allowing to serve content adapted to mobile devices, new ways to exchange information between users could further improve the attractiveness of mobile computing. Due to the limited capacity of mobile devices and the growing amount of data consumed and produced, cloud computing, as a solid and scalable foundation, has the assets to concretize the promising future of mobile computing.

Bibliography

[1] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2/.

[2] Amazon simple storage service (amazon s3). http://aws.amazon.com/s3/.

[3] Amazon.com. http://www.amazon.com.

[4] The apache cassandra project. http://cassandra.apache.org/.

[5] Berkeley db. http://www.oracle.com/technetwork/database/berkeleydb.

[6] Cleversafe. http://www.cleversafe.com/.

[7] Cloud afs. http://www.gladinet.com/.

[8] Cloudera flume. https://github.com/cloudera/flume.

[9] Common internet file system (cifs). http://msdn.microsoft.com/en-us/library/aa365233(VS.85).aspx.

[10] Couchdb. http://incubator.apache.org/couchdb/.

[11] dbshards. http://www.dbshards.com/dbshards/.

[12] Emc atmos. http://atmosonline.com/.

[13] The facebook. http://www.facebook.com.

[14] Facebook scribe. https://github.com/facebook/scribe.

[15] Geniedb. http://www.geniedb.com/.

[16] Google storage. http://code.google.com/apis/storage/.

[17] Hadapt. http://www.hadapt.com/.

[18] Hbase. http://hadoop.apache.org/hbase/.

[19] Ibm db2. http://www.ibm.com/software/data/db2/.

[20] Internet small computer systems interface (iscsi). http://tools.ietf.org/html/rfc3720.

[21] The iotop command. http://packages.debian.org/stable/iotop.

[22] Java platform, enterprise edition. http://www.oracle.com/technetwork/java/javaee/index.html.

[23] jclouds. http://www.jclouds.org/.

[24] libcloud. http://incubator.apache.org/libcloud/.

[25] The lsof command. http://packages.debian.org/stable/lsof.

[26] Memcached. http://memcached.org/.

[27] Microsoft azure. http://www.microsoft.com/windowsazure/.

[28] Microsoft sql server. http://www.microsoft.com/sqlserver/.

[29] Mogilefs. http://www.danga.com/mogilefs/.

[30] Mongodb. http://www.mongodb.org/.

[31] Mysql. http://mysql.com/.

[32] Nasuni filer. http://www.nasuni.com/.

[33] .net framework. http://www.microsoft.com/net.

[34] Nethogs: Net top tool grouping bandwidth per process. http://packages.debian.org/stable/nethogs.

[35] Network time protocol version 4: Protocol and algorithms specification. http://tools.ietf.org/html/rfc5905.

[36] Newsql. http://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/.

[37] Nimbusdb. http://nimbusdb.com/.

[38] Opentsdb. http://opentsdb.net/.

[39] Oracle database. http://www.oracle.com/us/products/database/index.html.

[40] Project voldemort. http://project-voldemort.com/.

[41] Rackspace cloud servers. http://www.rackspace.com/cloud/.

[42] Rackspace cloudfiles. http://www.rackspace.com/cloud/.

[43] Scalability best practices at ebay. http://www.infoq.com/articles/ebay-scalability-best-practices.

[44] Scalebase. http://www.scalebase.com/.

[45] Schooner. http://www.schoonerinfotech.com/.

[46] Scidb. http://www.scidb.org/.

[47] Simple object access protocol (soap). http://www.w3.org/TR/soap/.

[48] sysstat: System performance tools for linux. http://packages.debian.org/stable/sysstat.

[49] The top command. http://www.debian.org/doc/manuals/debian-reference/ch09.en.html#_the_top_command.

[50] Twitter. http://www.twitter.com.

[51] Voltdb. http://voltdb.com/.

[52] Wuala. http://www.wuala.com/.

[53] Xeround. http://xeround.com/.

[54] D. Abadi. Problems with cap, and yahoo's little known nosql system, 2010. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.

[55] M. L. Abbott and M. T. Fisher. *Scalability Rules: 50 Principles for Scaling Web Sites*. Pearson Education, Limited, 2011.

[56] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/g: Killer application for the global grid? In *Proc. of the IPDPS*, 2000.

[57] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. Racs: A case for cloud storage diversity. In *ACM SOCC*, Indianapolis, USA, 2010.

[58] M. Afergan, J. Wein, and A. LaMeyer. Experience with some principles for building an internet-scale reliable system. In *Proceedings of the 2nd conference on Real, Large Distributed Systems - Volume 2*, WORLDS'05, pages 1–6, Berkeley, CA, USA, 2005. USENIX Association.

[59] D. Agrawal and A. E. Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB '90: Proc. of the 16th International Conference on Very Large Data Bases*, pages 243–254, Brisbane, Queensland, Australia, 1990.

[60] Y. Al-Houmaily and P. Chrysanthis. Two-phase commit in gigabit-networked distributed database. In *Proc. of the Parallel and Distributed Computing Systems*, Orlando, Florida, USA, 1995.

[61] F. attern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[62] L. Badger, T. Grance, R. Patt-Corner, and J. Voas. Nist's cloud computing synopsis and recommendations. `http://csrc.nist.gov/publications/drafts/800-146/Draft-NIST-SP800-146.pdf`.

[63] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, 1984.

[64] P. A. Bernstein and N. Goodman. A proof technique for concurrency control and recovery algorithms for replicated databases. *Distributed Computing*, pages 32–44, 1987.

[65] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[66] N. Bonvin, T. G. Papaioannou, and K. Aberer. Dynamic cost-efficient replication in data clouds. In *Proc. of the Workshop on Automated Control for Datacenters and Clouds*, Barcelona, Spain, June 2009.

[67] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proc. of the 16th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2009.

[68] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[69] J. Brunelle, P. Hurst, J. Huth, L. Kang, C. Ng, D. C. Parkes, M. Seltzer, J. Shank, and S. Youssef. Egg: an extensible and economics-inspired open grid computing platform. In *Proc. of the GECON*, Singapore, May 2006.

[70] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[71] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. of the Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 2006.

[72] B-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of the NSDI*, 2006.

[73] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[74] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. of the SOSP*, 2001.

[75] M. Dahlin, B. Baddepudi, V. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2):300–313, 2003.

[76] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM System Journal*, 43(1):136–158, 2004.

[77] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB '06: Proc. of the 32nd international conference on Very large data bases*, pages 715–726, Seoul, Korea, 2006.

[78] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2007.

[79] J. Dejun, G. Pierre, and C. H. Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Proc. of NFPSLAM-SOC*, Stockholm, Sweden, 2009.

[80] C. Dellarocas. Goodwill hunting: An economically efficient online feedback mechanism for environments with variable product quality. In *Proc. of the Workshop on Agent-Mediated Electronic Commerce*, July 2002.

[81] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[82] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *IEEE Transactions on Information Theory*, 2010.

[83] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Transparent symmetric active/active replication for service-level high availability. In *Proc. of the CCGrid*, 2007.

[84] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[85] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Theoretical Computer Science*, 1992.

[86] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[87] B. Fitzpatrick. Livejournal's backend: A history of scaling. In *OSCON*, 2005.

[88] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[89] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Sigact News*, 33:51–59, 2002.

[90] R. A. Golding. Weak-consistency group communication and membership (ph.d. dissertation). Technical report, Santa Cruz, CA, USA, 1992.

[91] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[92] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[94] R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity, 2003.

[95] R. G. Guy, J. S. Heidemann, and Jr. T. W. Page. The ficus replicated file system. *ACM SIGOPS Operating Systems Review*, 26(2):26, 1992.

[96] D. Pisinger H. Kellerer, U. Pferschy. *Knapsack Problems*. Springer Verlag, 2004.

[97] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, December 1983.

[98] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 18:1–18:12, Berkeley, CA, USA, 2007. USENIX Association.

[99] J. Hamilton. One size does not fit all, 2009. `http://perspectives.mvdirona.com/CommentView,guid,afe46691-a293-4f9a-8900-5688a597726a.aspx`.

[100] G. Heiser, F. Lam, and S. Russell. Resource management in the mungi single-address-space operating system. In *Proc. of Australasian Computer Science Conference*, Perth, Australia, February 1998.

[101] A. Helsinger and T. Wright. Cougaar: A robust configurable multi agent platform. In *Proc. of the IEEE Aerospace Conference*, 2005.

[102] C. Henderson. *Building Scalable Web Sites: Building, Scaling, and Optimizing the Next Generation of Web Applications*. O'Reilly Media, Inc., 2006.

[103] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[104] M. Hofmann and L. R. Beaumont. *Content Networking: Architecture, Protocols, and Practice (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[105] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proc. of Int. Symposium on Parallel and Distributed Processing*, Nice, France, April 2003.

[106] S. Hopkins and B. Coile. Aoe (ata over ethernet), 2009. `http://support.coraid.com/documents/AoEr11.txt`.

[107] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[108] k. Birman. The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41:8–13, October 2007.

[109] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting cost amortization for query services. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 325–336, New York, NY, USA, 2011. ACM.

[110] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[111] B. Keating. Challenges involved in multimaster replication, 2001. `http://www.dbspecialists.com/files/presentations/mm_replication.html`.

[112] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[113] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[114] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[115] L. Lamport. Future directions in distributed computing. chapter Lower bounds for asynchronous consensus, pages 22–23. Springer-Verlag, Berlin, Heidelberg, 2003.

[116] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[117] N. Laranjeiro and M. Vieira. Towards fault tolerance in web services compositions. In *Proc. of the workshop on engineering fault tolerant systems*, New York, NY, USA, 2007.

[118] H. Li, M. Muskulus, and L. Wolters. Modeling job arrivals in a data-intensive grid. In *Proc.12 th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 210–231. Springer, 2006.

[119] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. Sla-driven clustering of qos-aware application servers. In *IEEE Trans. Softw. Eng*, 2007.

[120] S. K. Madria and B. Bhargava. A transaction model for mobile computing. In *Proceedings of the 1998 International Symposium on Database Engineering & Applications*, pages 92–, Washington, DC, USA, 1998. IEEE Computer Society.

[121] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, pages 12–17, New York, NY, USA, 2008. ACM.

[122] P. Mell and T. Grance. Nist's cloud computing definition. `http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc`.

[123] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, 1988. Springer-Verlag.

[124] T. N. Minh and L. Wolters. Modeling job arrival process with long range dependence and burstiness characteristics. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 324–330, Washington, DC, USA, 2009. IEEE Computer Society.

[125] J. Norris, K. Coleman, A. Fox, and G. Candea. Oncall: Defeating spikes with a free-market application cluster. In *Proc. of the International Conference on Autonomic Computing*, New York, NY, USA, May 2004.

[126] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44:2–19, August 2010.

[127] C. Pautasso, T. Heinis, and G. Alonso. Autonomic resource provisioning for software business processes. *Information and Software Technology*, 49:65–80, 2007.

[128] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: replicated database services for world-wide applications. In *Proc. of the 7th workshop on ACM SIGOPS European workshop*, Connemara, Ireland, 1996.

[129] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[130] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of 5th USENIX Conference on File and Storage Technologies (FAST'07)*, San Jose, CA, USA, February 2007.

[131] E. Pitoura and B. Bhargava. Data consistency in intermittently connected distributed systems. *IEEE Trans. on Knowl. and Data Eng.*, 11:896–915, November 1999.

[132] O. Regev and N. Nisan. The popcorn market. online markets for computational resources. *Decision Support Systems*, 28(1-2):177–198, 2000.

[133] C. Reich, K. Bubendorfer, M. Banholzer, and R. Buyya. A sla-oriented management of containers for hosting stateful web services. In *Proc. of the IEEE Conference on e-Science and Grid Computing*, Washington, DC, USA, 2007.

[134] R. Rivest. The md5 message-digest algorithm, 1992.

[135] H. Robinson. Cap confusion: Problems with 'partition tolerance', 2010. `http://www.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/`.

[136] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *International Workshop IPTPS*, 2005.

[137] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, 2001.

[138] M. Rys. How do large-scale sites and applications remain sql-based?, 2011. `http://queue.acm.org/detail.cfm?id=1971597`.

[139] J. Salas, F. Perez-Sorrosal, N.-M. M. Pati, and R. Jiménez-Peris. Ws-replication: a framework for highly available web services. In *Proc. of the WWW*, pages 357–366, New York, NY, USA, 2006.

[140] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation or the sun network filesystem. 1985.

[141] G. Sanders and S. Shin. Denormalization effects on performance of rdbms. *Hawaii International Conference on System Sciences*, 3:3013, 2001.

[142] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *Transactions on Computers*, 39(4):447–459, 1990.

[143] T. Schlossnagle. *Scalable Internet Architectures*. 2006.

[144] M. Shapiro, P. Dickman, and D. Plainfossè. Robust, distributed references and acyclic garbage collection. In *Proc. of the Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1992.

[145] S. K. Shin and G. L. Sanders. Denormalization strategies for data retrieval from data warehouses. *Decis. Support Syst.*, 42:267–282, October 2006.

[146] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed systems. *IEEE Transactions on Software Engineering*, pages 219–228, 1983.

[147] H. Stockinger, K. Stockinger, E. Schikuta, and I. Willers. Towards a cost model for distributed and replicated data stores. In *Proc. of Euromicro Workshop on Parallel and Distributed Processing*, Italy, February 2001.

[148] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[149] M. Stonebraker. In search of database consistency. *Commun. ACM*, 53:8–9, October 2010.

[150] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in mariposa. In *Proc. of Parallel and Distributed Information Systems*, Austin, TX, USA, September 1994.

[151] C. Strauch. Nosql databases. pages 1–149, 2010.

[152] N. Ntarmos T. Pitoura and P. Triantafillou. Replication, load balancing and efficient range query processing in dhts. In *Proc. of Int. Conference on Extending Database Technology*, Munich, Germany, March 2006.

[153] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4:180–209, June 1979.

[154] Jinesh Varia. Architecting for the cloud: Best practice. 2010.

[155] W. Vogels. Eventually consistent, December 2008.

[156] M. N. Dailey W. Iqbal and D. Carrera. Sla-driven dynamic resource management for multi-tier web applications in a cloud. In *Proc. of the CCGrid*, Melbourne, Australia, 2010.

[157] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18:103–117, 1992.

[158] M. Wang and T. Suda. The bio-networking architecture: a biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications. In *Proc. of the IEEE Symposium on Applications and the Internet*, 2001.

[159] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from IPTPS'01*, 2002.

[160] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. Van Steen. Service-oriented data denormalization for scalable web applications. In *Proceedings of the 17th International World Wide Web Conference*, Beijing, China, April 2008.

[161] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[162] D. Weinreb. What does the proof of the "cap theorem" mean?, 2010. `http://danweinreb.org/blog/what-does-the-proof-of-the-cap-theorem-mean`.

[163] D. Weinreb. Improving the pacelc taxonomy, 2011. `http://danweinreb.org/blog/improving-the-pacelc-taxonomy`.

Curriculum Vitae

Nicolas Bonvin

Contact information

Birthdate	01 april 1980
Nationality	swiss
email	nbonvin @ niin.org

Work Experience

LSIR, IC, EPFL, Lausanne, Switzerland **2007 – 2011**

PhD Candidate

- Research area : cloud computing, NoSQL databases, scalability and high-availability of distributed systems, large-scale systems, web of entities, P2P
- Development of P-Grid, a fully decentralized P2P network
- Participation in European projects (NEPOMUK, OKKAM)

CTP (a Novell business), Geneva, Switzerland **2005 – 2007**

Consultant, Novell Linux engineer

- Secure email communication, IncaMail, Swiss Post, Switzerland
- Technical Support at Nestle, Vevey, Switzerland
- Document Management at State of Vaud, Lausanne, Switzerland
- Identity Management at IMD, Lausanne, Switzerland
- Business Process Management at SITA, Geneva, Switzerland

- Infrastructure deployment and management

ProLibre Sarl, Geneva, Switzerland **2004 – 2005**

Developer, System adiministrator

- Development of a secure e-voting system
- Administration of Linux servers

Skutale Technologies, Fribourg, Switzerland **2003 – 2010**

Co-Founder, CTO

- Multi-Datacenter Managed Hosting (Web, Email, DNS, Applications)
- Web Development

Education

EPFL, Lausanne, Switzerland **2005**

Master of Science in Computer Science

- Ing. inf. dipl. EPF
  Thesis : "E-Voting System"
  Advisor : Prof. Serge Vaudenay

Kollegium Spiritus Sanctus, Brig, Switzerland **1999**

College degree

- Maturite Cantonale Type B (Latin-Anglais)

Selected Publications

N. Bonvin, T. G. Papaioannou, and K. Aberer. "Autonomic SLA-driven Provisioning for Cloud Applications". In 11th *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

N. Bonvin, T. G. Papaioannou, and K. Aberer. "An economic approach for scalable and highly-available distributed applications". In Proc. of the 3rd *IEEE International Conference on Cloud Computing (CLOUD)*, 2010.

Z. Miklos, N. Bonvin, P. Bouquet, M. Catasta, D. Cordioli, P. Fankhauser, J. Gaugaz, E. Ioannou, H. Koshutanski, A. Mana, C. Niederee, T. Palpanas, and H. Stoermer. "From Web Data to Entities and Back". In 22nd *International Conference on Advanced Information Systems Engineering (CAISE)*, 2010.

N. Bonvin, T. G. Papaioannou, and K. Aberer. “A self-organized, fault-tolerant and scalable replication scheme for cloud storage”. In Proc. of the *ACM Symposium on Cloud Computing 2010 (SOCC)*, 2010.

N. Bonvin, T. G. Papaioannou, and K. Aberer. “Cost-efficient and Differentiated Data Availability Guarantees in Data Clouds”. In the 26th *IEEE International Conference on Data Engineering (ICDE)*, 2010.

E. Ioannou, S. Sathe, N. Bonvin, A. Jain, S. Bondalapati, G. Skobeltsyn, C. Niederée, and Z. Miklos. “Entity Search with NECESSITY”. In Proceedings of the 12th *International Workshop on the Web and Databases*, 2009.

N. Bonvin, T. G. Papaioannou, and K. Aberer. “Dynamic Cost-Efficient Replication in Data Clouds”. In First *ACM Workshop on Automated Control for Datacenters and Clouds (ACDC)*, 2009.

Technical Skills

Database

· MySQL · BDB · Oracle · NoSQL (Cassandra, Voldemort, Redis, MongoDB, HBase, Memcached) · Hadoop

Programming

· Java · C++ · PHP · Perl · SQL · Scala · JavaScript · HTML · Shell

Operating Systems

· Linux · Unix (*BSD, Solaris) · Windows

Miscellaneous

· Security · Cryptography · High-Availability · Scalability · Distributed Systems · Large-Scale Systems · P2P · Cloud Computing

Additional Information

Languages

· English : fluent

· German : good knowledge

· Swiss German : good knowledge

· French : mother tongue

Interests

· Paragliding · Ski Freeride · Mountain · Krav Maga