BlobSeer as a Data-Storage Facility for Clouds: Self-adaptation, Integration, Evaluation

Alexandra Carpen-Amarie

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention : Informatique

École doctorale MATISSE

présentée par

Alexandra Carpen-Amarie

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

BlobSeer as a data-storage facility for Clouds: self-adaptation, integration, evaluation

Thèse soutenue le 15 décembre 2011
devant le jury composé de :

Christian PEREZ ,
Directeur de recherche - INRIA Grenoble Rhône-Alpes / *rapporteur*
María S. PÉREZ-HERNÁNDEZ
Professeur des universités - Université Polytechnique de Madrid / *rapporteur*

Adrien LÈBRE
Chargé de recherche - École des Mines de Nantes / *examinateur*
Nicolae TAPUS
Professeur des universités - Université Polytechnique de Bucarest / *examinateur*

Luc BOUGÉ
Professeur des universités - ENS Cachan-Bretagne / *directeur de thèse*
Gabriel ANTONIU
Chargé de recherche - INRIA Rennes Bretagne-Atlantique / *directeur de thèse*

Contents

Chapter 1

Introduction

Contents

Cloud computing emerged as a promising paradigm for hosting and delivering on-demand services on a pay-per-use basis. This model builds on widely popular technologies, such as *Grid computing*, *Utility computing* or *Internet services computing*, aiming at making better use of distributed resources and providing high-performance, scalable and cost- effective services. The Cloud computing model has drawn the interest of both the academic community and large companies, such as Amazon, Google or Microsoft, which strive to address the new challenges that surfaced as Cloud computing became an attractive solution for a broad spectrum of applications.

One of the main challenges in this context is data management, as a result of the exponential growth of the volume of data processed by distributed applications nowadays. The need for data-management solutions is particularly critical in the area of scientific applications. For instance, in the area of high- energy physics, CERN's Large Hadron Collider experiments involving the CMS detector are expected to generate over 15 petabytes of data per year.

Several storage systems have been proposed to address the increasing demands for scalable and efficient data management, some of them emerging from the data-intensive distributed computing community, while others have been specifically designed for Cloud environments. Nevertheless, some requirements for massive data management at global scales have still to be dealt with.

An essential concern in this case is the complexity of building advanced systems at such scales. Thus, as more evolved systems are being designed, the high degree of complex-

ity related to their configuration and tuning is becoming a limiting factor that threatens to overwhelm the current management capabilities and render the systems unmanageable and insecure. To overcome such challenges, a system can be outfitted with a set of self- management mechanisms that enable an autonomic behavior, which can shift the burden of understanding and managing the system's state from the human administrator to an automatic decision-making engine.

Cloud-oriented systems also face a different type of issues, arising from the multitude of service providers that have entered the Cloud computing market. To process massive amounts of data, interoperability between various Cloud providers and services turns up to be a major challenge that restricts the resources to which a user has access to the data centers of a specific provider. To overcome such boundaries, Cloud providers need to converge towards compatible standards.

1.1 Contributions

The goal of this thesis is to enhance a distributed data-management system with self-management capabilities, so that it can meet the requirements of the Cloud storage services in terms of scalability, data availability, reliability and security.

Furthermore, we investigate the requirements of Cloud data services in terms of data-transfer performance and access patterns and we explore the ways to leverage and adapt existing data-management solutions for Cloud workloads. We aim at building a Cloud data service both compatible with state-of-the-art Cloud interfaces and able to deliver high-throughput data storage.

The contributions of this thesis can be summarized as follows:

Introspection mechanisms for data-management systems. Introspection is the prerequisite of an autonomic behavior, the first step towards performance improvement and resource-usage optimization for data-management systems distributed at large scales. In order to enable a data storage platform with introspection capabilities we have designed a three-layered architecture aiming at identifying and generating relevant information related to the state and the behavior of the system. The work further focused on enhancing BlobSeer, a distributed storage system designed for highly-efficient concurrent data accesses, with self-awareness capabilities based on the previously introduced introspection architecture.

Additionally, we designed a visualization tool to provide graphical representations of the data collected and processed by the BlobSeer introspection component. This work was carried out in collaboration with Jing Cai, a Master student from the City University in Hong Kong.

A generic framework for enforcing security policies in storage systems. We proposed a generic security management framework to enable self-protection for data-management systems. We consider several security challenges that can impact on Cloud data services and we provide a flexible framework for security policies definition and enforcement to address them. As a case study, the proposed framework was applied to BlobSeer and was evaluated in the context of Denial of Service attacks. The

obtained results showed that our self-protection architecture meets the requirements of a data storage system in a large-scale deployment: it was able to deal with a large number of simultaneous attacks and to restore and preserve the performance of the target system. This work was carried out in collaboration with Cristina Basescu, Catalin Leordeanu and Alexandru Costan from the Politehnica University of Bucharest, Romania, within the framework of the "DataCloud@work" INRIA Associate Team.

Self-configuration in data-management systems through dynamic dimensioning. We addressed the problem of accurately estimating the most advantageous deployment configuration of a data management system in terms of number of storage nodes. In this context, we proposed a dynamic configuration framework designed to work in conjunction with the introspection mechanisms to automatically adjust the number of deployed storage nodes. We validated the proposed framework within a integration with BlobSeer, for which we implemented a self-configuration component able to contract and expand the pool of storage nodes based on configurable monitoring parameters, such as the rate of data accesses or the load of the storage nodes. This work was carried out in collaboration with Lucian Cancescu, Alexandru Palade and Alexandru Costan from the Politehnica University of Bucharest, Romania, within the framework of the "DataCloud@work" INRIA Associate Team.

A BlobSeer-based file system as a storage backend for Cloud services. We investigated the challenges of providing high-throughput data storage in Cloud environments. We targeted two directions: data management for virtual machine images and for Cloud application data. Our contribution lies in the design and implementation of a BlobSeer-based file system optimized to efficiently serve as a storage backend for Cloud services. We validated our proposed file system by integrating it within a real-world Cloud environment, the Nimbus platform. Large-scale synthetic experiments have been performed to assess the performance of our customized distributed Cloud storage service. The benefits and drawbacks of using Cloud storage for real-life applications have been emphasized in evaluations that involved data-intensive MapReduce applications and tightly-coupled, high-performance computing applications. This work was carried out under the supervision of Kate Keahey and John Bresnahan during a 2-month internship at Argonne National Laboratory, Chicago, Illinois, USA.

All experiments performed in the context of the aforementioned contributions were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

1.2 Publications

Journals:

- Carpen-Amarie Alexandra, Costan Alexandru, Leordeanu Catalin, Basescu Cristina, Antoniu Gabriel, **Towards a Generic Security Framework for Cloud Data Management Environments.** To appear in *International Journal of Distributed Systems and Technologies*, 2012.

- Carpen-Amarie Alexandra, Costan Alexandru, Cai Jing, Antoniu Gabriel, Bougé Luc, **Bringing Introspection into BlobSeer: Towards a Self-Adaptive Distributed Data Management System.** In *International Journal of Applied Mathematics and Computer Science*, Vol. 21, No. 2, pages 229-242, 2011.

- Nicolae Bogdan, Antoniu Gabriel, Bougé Luc, Moise Diana, Carpen-Amarie Alexandra, **BlobSeer: Next Generation Data Management for Large Scale Infrastructures.** In *Journal of Parallel and Distributed Computing*, Vol. 71, No. 2, pages 168-184, 2011.

Conferences and workshops:

- Carpen-Amarie Alexandra, **Towards a Self-Adaptive Data Management System for Cloud Environments.** In *IPDPS 2011: 25th IEEE International Parallel and Distributed Processing Symposium: PhD Forum*, pages 2072-2075, Anchorage, USA, 2011.

- Moise Diana, Carpen-Amarie Alexandra, Antoniu Gabriel, Bougé Luc, **A Cost-Evaluation of MapReduce Applications in the Cloud.** To appear in the *Grid'5000 Spring School Proceedings*, Reims, France, 2011.

- Basescu Cristina, Carpen-Amarie Alexandra, Leordeanu Catalin, Costan Alexandru, Antoniu Gabriel, **Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies.** In *AINA 2011: 25th International Conference on Advanced Information Networking and Applications*, pages 459-466, Singapore, 2011.

- Carpen-Amarie Alexandra, Cai Jing, Costan Alexandru, Antoniu Gabriel, Bougé Luc, **Bringing Introspection Into the BlobSeer Data-Management System Using the MonALISA Distributed Monitoring Framework.** In *ADiS 2010: International Conference on Complex, Intelligent and Software Intensive Systems, Workshop on Autonomic Distributed Systems*, pages 508-513, Krakow, Poland, 2010.

- Carpen-Amarie Alexandra, Andreica Mugurel, Cristea Valentin, **An Algorithm for File Transfer Scheduling in Grid Environments.** In *HiPerGrid 2008: International Workshop on High Performance Grid Middleware*, pages 33-40, Bucharest, Romania, 2008.

Research reports:

- Alexandra Carpen-Amarie, Tuan Viet Dinh, Gabriel Antoniu, **Efficient VM Storage for Clouds Based on the High-Throughput BlobSeer BLOB Management System**, INRIA Research Report No. 7434, INRIA, Rennes, France, 2010.

- Alexandra Carpen-Amarie, Jing Cai, Luc Bougé, Gabriel Antoniu, Alexandru Costan, **Monitoring the BlobSeer distributed data-management platform using the MonALISA framework**, INRIA Research Report No. 7018, INRIA, Rennes, France, 2009.

1.3 Organization of the manuscript

This manuscript is organized into five parts.

The first part discusses the context of our work. Chapter 2 introduces the Cloud computing paradigm, focusing on Infrastructure-as-a-Service Clouds. The chapter closes on the research issues that emerged as Cloud computing developed into a popular paradigm for both the industry and the scientific community. Among these challenges, we further concentrate on data management. Thus, Chapter 3 provides a survey of some of the most widely used distributed storage systems. We study parallel file systems, specialized file systems for data-intensive applications, as well as Cloud storage systems, analyzing the properties required for providing Cloud services that efficiently handle large amounts of data. Next, we focus on exploiting self-management mechanisms to optimize such file systems. To this end, Chapter 4 discusses a set of self-* properties and their existing implementations at the level of data-management systems. Finally, this part closes with Chapter 5, which introduces BlobSeer, a high-performance, large-scale distributed storage service we used as a case study throughout this manuscript.

The second part presents the first contribution of this thesis, namely a set of frameworks designed to enhance data-management systems with self-* properties. Chapter 6 introduces three generic approaches for self-management, aiming at defining self-awareness, self-protection and self-configuration components for distributed data storage systems. In Chapter 7, we validate the aforementioned self-* frameworks by integrating them into BlobSeer. We discuss implementation details and the specific interfacing modules required to enable BlobSeer with self-management components. The last chapter of this part, namely Chapter 8, introduces our large-scale experimental testbed, the Grid'5000 platform, together with a set of deployment tools we developed to evaluate our proposed frameworks. Then, we present the experiments we conducted to assess the benefits of the self-* mechanisms in BlobSeer.

The third part introduces the Nimbus Cloud framework and focuses on Cumulus, the data storage service that accompanies the Nimbus Infrastructure-as-a-Service Cloud. Next, Chapter 10 presents our contribution with respect to Cloud data storage. We designed a BlobSeer-based file system to play the role of the backend storage layer for Cumulus. We close this part with a set of benchmarks that evaluate the performance and scalability of the Cumulus storage service and the Blobseer-based backend.

The fourth part consists of two chapters that present various evaluations of the BlobSeer-based Cloud storage service in the context of real-life applications. Chapter 11 investigates performance- and cost-related aspects of executing MapReduce applications in Cloud environments. In Chapter 12, we analyze the impact of using Cloud-based storage systems for tightly-coupled applications, such as CM1, a simulator for modeling atmospheric phenomena. Both chapters include experiments conducted on hundreds of nodes in Grid'5000.

The fifth part summarizes the contributions of the thesis in Chapter 13 and discusses a series of unexplored research challenges revealed by our work in Chapter 14.

Part I

Context: data storage for Cloud environments

Chapter 2

Cloud computing

Contents

The main concepts that drive the development of Cloud computing have emerged back in the 1960s, when Douglas Parkhill investigated the principles of "utility computing" [86]. The idea behind "utility computing" comes from the analogy with the traditional public utilities, such as the *electricity grid*, which allows consumers to obtain electric power on-demand, while offering the illusion of infinite resources available for an unlimited number of clients.

Initially, the idea of providing computational resources on-demand has been developed into the *Grid computing* paradigm, which aimed at coordinating resources distributed over various administrative domains to solve large computational problems. The term "grid" reflects the analogy with the electricity network, suggesting that Grid infrastructures should deliver computational power to any user, regardless of the details of the underlying resources or the problem complexity. As Grid technologies gained an increasing popularity in the academic community, standard protocols, middleware and services have been proposed to address the various requirements of scientific applications. In this context, many research efforts concentrated on providing a Grid definition [34, 35], among which the most widely accepted is the one proposed by Ian Foster in 2002: "a system that coordinates re-

sources which are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service" [32].

Cloud computing shares the same vision as the Grid paradigm: it aims at federating distributed computational resources to deliver on-demand services to external customers. However, whereas most Grid platforms were targeting resource sharing for the scientific community, Cloud computing shifts the focus on a business model where the clients pay for the resources acquired on demand. Clouds leverage existing technologies, such as virtualization, to provide dynamic and flexible resource provisioning that addresses precise user needs through user-transparent web-based services.

2.1 Defining Cloud computing

Numerous definitions have been proposed for Cloud computing, focusing on various aspects that characterize the paradigm [109, 31, 37]. We consider the definition of Cloud computing proposed by The National Institute of Standard and Technology (NIST) [67], as it illustrates the essential facets of the Cloud:

> « *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.* »

Several authors [31, 44, 116] highlight the features that make the Cloud computing paradigm attractive to both the commercial and academic communities. We summarize the key aspects that played an important role in the increasing popularity of Clouds.

Virtualized resources. Virtualization provides an abstraction layer hiding the complexity and heterogeneity of the physical resources that make up the Cloud. Recent virtualization technologies, such as KVM [60] or Xen [10], have reduced the performance penalty of running applications in virtualized environments. Thus, Clouds widely adopted virtualization, as a solution for server consolidation and resource availability.

Scalability. Cloud services aggregate large pools of physical resources belonging to the same data center or federating resources from multiple providers. In order to provide reliable services, Cloud frameworks have to efficiently take advantage of the lower-level services and underlying physical machines and to transparently scale to handle rapid increase in service demands.

Ease of access. The Cloud computing paradigm features two important properties to attract both commercial and academic actors. First, it allows users to rent computational or storage resources for the needs of their applications, instead of requiring them to invest in the infrastructure. Furthermore, leased resources can dynamically scale up and down on the fly, according to the real-time load. The user can access its resources from anywhere in the world, and the Cloud provider guarantees to comply with a set of *service-level agreements* (SLAs) with respect to resource availability and provided quality of service.

Pay-per-use model. A key aspect of Cloud computing is the model of providing resources on demand and charging clients only for what they use. Adopting this economic model, Cloud providers can maximize resource utilization, while offering clients affordable services.

2.2 The Cloud computing landscape

Several surveys have focused on formulating a Cloud computing taxonomy. We focus on two types of classifications, taking into account the deployment model of the Cloud frameworks and the type of provided services.

2.2.1 Deployment models

Companies, as well as scientific organizations, have various needs with respect to the type, availability or security of the services they employ. To address them, several types of Clouds have been proposed, each of them presenting specific strengths and weaknesses.

Public Cloud. A *public Cloud* provides services to any client that has access to the internet. Typically, the services are managed by off-site providers, which also control the network and security settings, while the user can obtain on-demand resources billed on a pay-per-use basis.

Private Cloud. In this case, the infrastructure is exclusively used by a single organization. Its advantage is that it offers the highest degree of security and reliability. On the downside, having access to a *private Cloud* implies having full access to the physical infrastructure and also building and managing the Cloud.

Hybrid Cloud. *Hybrid Clouds* aim at addressing the limitations of the other deployment models. A combination between *private* and *public Clouds*, the hybrid model preserves the security guarantees of the first model, enhancing it with the possibility to dynamically scale by renting resources from *public* cloud providers when the computational needs overload the local resources. Achieving such a degree of flexibility is however a challenging task, as many Cloud services or interfaces are incompatible and cannot be accessed through the same client tools.

2.2.2 Cloud services stack

The Cloud services can be classified as a stack of service layers, where each layer is built upon services provided by the lower levels of the stack:

Infrastructure-as-a-Service (IaaS). This is the lowest level of the Cloud computing stack, supplying users with on-demand access to virtualized resources they can fully configure. Typically, Infrastructure-as-a-Service Cloud users can rent computational, storage or networking resources for predefined amounts of time. They have access to infrastructure management services such as automatic resource deployment and dynamic scaling of the number of leased nodes. Commercial Clouds feature a pay-per-use pricing scheme, which im-

plies low entry costs for clients, an important advantage in the case of customers who cannot afford buying their own infrastructure. The main actors on the commercial IaaS Cloud market include Amazon EC2 [8], GoGrid [39], Flexiscale [30]. The flexibility and wide range of potential applications of this service model have drawn the attention of the scientific community, who proposed several open-source IaaS solutions, such as Nimbus [57], OpenNebula [72], Eucalyptus [79] or OpenStack [83].

Platform-as-a-Service (PaaS). Cloud providers offer an additional abstraction layer built on top of IaaS functionalities. PaaS services consist in high-level integrated environments that enable users to build, test and execute their own applications. The purpose of PaaS environments is to hide the complexity of managing and deploying applications on top of virtual machines and enable users to focus only on the application logic. This approach comes, however, with a defining tradeoff: such services are not suitable for any type of application, as PaaS providers typically design specific environments and APIs through which the users can take advantage of the platform's capabilities. As an example, Amazon's Elastic MapReduce (EMR) [118] offers scalable and efficient data processing for massive datasets; nevertheless, the users are required to submit only applications complying with the MapReduce paradigm. Other popular PaaS commercial offerings include Microsoft Azure [90], Google Apps Engine [93], or Django [105], all of them being devised for hosting web applications. Not all these services provide the same degree of flexibility. Whereas platforms such as EMR provide specific programming environments, Microsoft Azure is an example of a platform that allows clients to employ a wider range of tools compatible with the offered execution environment.

Software-as-a-Service (SaaS). At the highest level of the hierarchy, SaaS delivers specialized software hosted in the Cloud, accessible for customers only through the Internet. Such applications are usually built on top of PaaS or IaaS Clouds. The main asset of SaaS applications is that they can rely on the powerful lower-level Cloud services to achieve better availability and performance by leveraging features like distributed processing or automatic scaling. Examples of SaaS applications are Google Maps [41] services, Google Docs [40] and Microsoft's Office Live [69].

2.3 Infrastructure-as-a-Service Clouds

In this section we focus on IaaS Clouds, as they offer the basic services on which all the other layers are built, as well as the highest degree of flexibility and control. We provide a survey of some state-of-the-art Cloud infrastructures, ranging from popular commercial Clouds to emerging open-source research projects:

Amazon Elastic Compute Cloud (EC2). Amazon EC2 [8] provides a large computing infrastructure, where the users can rent virtualized environments. It offers the illusion of infinite compute capacity, allowing users to dynamically resize the leased resources, while employing a pay-per-use business model. Typically, a user creates a virtual machine image, called Amazon Machine Image (AMI), containing all needed applications. Amazon provides

a broad set of tools to upload the image into the system and to launch multiple instances. An Amazon EC2 instance is a virtual machine booted from a specific AMI, relying on the Xen [10] virtualization engine. As it grants users full control of their instances, EC2 provides flexible environments that can be adapted to a multitude of applications, ranging from Internet services to scientific applications. Even though the architecture and implementation of the Amazon services is proprietary, the interaction with the client is done through public APIs that have become extremely popular and have been adopted by a large number of open-source projects.

Nimbus. The Nimbus infrastructure Cloud [57, 88] is an open-source IaaS implementation devised for the computational needs of the scientific community. It is interface-compatible with Amazon EC2, to enable users to switch from one Cloud offering to another without modifying their access tools. The Nimbus project provides as well a set of additional tools targeted to scientific applications. Thus, it includes mechanisms to create configurable virtual clusters, interoperability tools to explore Cloud federation and efficient virtual machine management components to support simultaneous deployment for hundreds of virtual machines.

OpenNebula. OpenNebula [72, 82] provides fully open-source IaaS services designed to address the requirements of business use cases across multiple industries, such as web hosting, telecommunications, eGovernment. It consists in a set of virtualization tools for managing local data centers, but also for interconnecting multiple Cloud environments. The main design principles on which the OpenNebula project relies include a modular and extensible architecture, scalability for large-scale infrastructures, interoperability with existing Cloud offerings, open-source implementation. Furthermore, OpenNebula aims at providing standardized interfaces for managing virtual machines and data, such as the Open Grid Forum (OGF) [81] OCCI [80] API or the Amazon EC2 "de-facto" industry standard.

Eucalyptus. The Eucalyptus [79, 87] project aims at addressing several important requirements in IaaS Clouds: extensibility, scalability, high availability, elastic virtual machine pool reconfiguration. It is fully-compatible with Amazon's EC2 interface and it includes robust tools for creating hybrid Clouds based on EC2 or EC2-compatible IaaS offerings. Eucalyptus is one of the most widely used IaaS solutions for private data centers. To meet the needs of the enterprise environments, the Eucalyptus Cloud is equipped with secure services for user management and resource quotas, as well as with extensive tools for controlling VMs lifecycle.

OpenStack. OpenStack Compute [83] is an open-source toolkit designed to provision and manage large networks of virtual machines. It can be integrated with a variety of hardware configurations and hypervisor, storage and networking solutions. Targeted towards the needs of industry projects, it provides scalable and reliable services suitable for a broad class of data centers. Furthermore, it defines specific APIs to allow users to easily and securely create, launch, and interact with their instances.

2.4 Research challenges

Cloud computing is an active research area, which is continuously evolving towards new types of services and new application models. Cloud technologies face different challenges at each level of the Cloud computing stack. Some key aspects that impact a wide adoption rate of IaaS Clouds, in particular in the context of scientific applications, are detailed below:

Efficient data management. Large-scale data-intensive applications typically rely on dedicated distributed file systems. Such file systems however do not expose traditional APIs or POSIX semantics. Furthermore, they are not suitable for Cloud environments, being incompatible with the access and cost model employed in such contexts. On the other hand, Cloud data services introduce compatibility challenges with respect to existing applications. Research efforts in this area aim at finding appropriate data storage models that address the data-intensive applications needs and comply with Cloud environments restrictions.

Elasticity. Cloud computing is based on the capability to provide on-demand resources. Furthermore, the pool of rented resources has to scale up as the application load increases, and to automatically shrink when the peak processing load has been overcome, to minimize user costs. Elastic services need to provide an acceptable tradeoff, avoiding performance penalties, as well as ineffective resource usage. In this context, the challenge lies in identifying specific mechanisms to allow an automatic correlation between the state of the system and the lease dimension.

Security. This is an essential research topic for Cloud computing. The security issues that arise in Cloud environments are twofold. First, data security greatly impacts the applications that want to rely on Cloud services for storing or processing sensitive data. The drawback in this case is that the user has no control over the security policies and the protection mechanisms for its data. Second, Cloud providers are also confronted with security issues, as they need to counterbalance malicious attacks in order to sustain a constant service availability and performance. As a consequence, Cloud providers have to implement efficient detection mechanisms to protect and repair their systems in case of security attacks.

Interoperability. Many Cloud computing offerings have evolved as independent frameworks that provide similar user services through specific interfaces. The inner management techniques inside the Cloud are transparent to the user, which benefits from specific tools to access the services. However, the users may face interoperability issues if they attempt to switch to another Cloud provider. These challenges relate to the lack of standardized access protocols and interfaces across Cloud providers. This also poses an additional constraint on Cloud federation attempts, which have to consider the various access models to succeed in interconnecting several Cloud platforms.

2.5 Summary

In this chapter, we consider the Cloud computing landscape, providing an insight on the various definitions of the Cloud paradigm and the existing service models. Further, we present a survey of the most widely employed Infrastructure-as-a-Service Cloud offerings. Finally, we investigate some of the most active research topics that impact the adoption scale of Cloud services.

Chapter 3

Scalable distributed storage systems

Contents

In this chapter we survey a set of distributed data-management solutions designed to provide storage support for various type of large-scale applications. We point out the specific design principles implemented by each systems, summarizing their limitations and new challenges at the end of the chapter.

3.1 Parallel file systems

The high-performance computing community proposed various distributed file systems that target scalability and high-throughput data management. Some of the most representative parallel file systems are presented below, together with their main features.

Parallel Virtual File System. PVFS [49, 101] is an open-source, scalable parallel file system designed to provide high performance for parallel applications. It is able to scale to a very large number of storage servers and clients by relying on a distributed architecture to avoid single points of contention. Thus, data and metadata are distributed across multiple servers, ensuring high throughput for large I/O and concurrent file

accesses. To achieve high performance for large data and multiple concurrent clients, PVFS does not implement the POSIX semantics. It optimizes concurrent accesses to non-overlapping blocks of the same file, but does not support concurrent updates on the same portion of a file. PVFS also avoids inefficient distributed locking mechanisms, defining instead a sequence of steps for each operation that maintains the file system directory hierarchy in a consistent state at any time.

IBM General Parallel File System. GPFS [94] is one of the most popular parallel file systems employed in many important supercomputing facilities in both commercial and scientific communities. Its design focuses on scalability and high-performance access for petabytes of data. GPFS stripes data in equally-sized blocks typically of 256 KB in size and distributes metadata across multiple servers to avoid bottlenecks and single points of failure. Furthermore, it ensures data availability through replication and recovery mechanisms in the case of failures. It implements a fine-grained locking scheme enabling concurrent clients to perform efficient updates to different parts of the same file, while preventing inconsistent overlapping accesses. Its sophisticated token-based scalable locking mechanism also allows GPFS to fully support POSIX semantics, both at the level of data and at the level of file metadata.

Ceph. To address some of the main challenges for storage systems, such as scalability, reliability and performance, the Ceph [112, 99] distributed file systems relies on several design principles. First, it distributes data across object storage devices (OSD) [68] to achieve an efficient management of data accesses, update serialization or replication. Second, Ceph decouples data and metadata management and employes a distributed metadata management scheme to enhance scalability. Unlike other file systems, Ceph also includes dynamic metadata servers, which are able to adapt to changing workloads by repartitioning the file system hierarchy in order to achieve load balancing. To improve reliability, data is replicated across multiple OSDs and replicas are automatically re-created on new disks upon failures. Ceph clients can access the file system through a variety of interfaces, including a POSIX-compliant client implementation.

Lustre. Lustre [26, 62] is a POSIX-compliant object-based parallel file system designed to scale to tens of thousands of nodes with petabytes of storage capacity. Its architecture separates metadata management from file data storage. Lustre employs a set of distributed metadata servers for filesystem operations, each of them relying on backend high-performance local file systems. Furthermore, file data is striped across thousands of Object Storage Servers, which manage block allocation and data access locking. The clients obtain the layout of each file from the metadata servers, and then they can directly access the storage servers, avoiding contention and obtaining a highly scalable I/O performance. Currently, Lustre is the backend file system for the first two supercomputers in Top500 [108].

3.2 Distributed file systems for data-intensive workloads

MapReduce [23] was introduced by Google as a paradigm that enables large-scale computations for massive datasets. The MapReduce programming model is based on two functions

specified by the user: *map* parses key/value pairs and passes them as input to the *reduce* function. MapReduce frameworks, such as Hadoop, take care of splitting the data, scheduling the tasks and executing them in parallel on multiple machines. In this context, specialized distributed file systems have been proposed to deal with specific access patterns that require support for highly concurrent and fine-grained access to data.

Google File System. To meet data-intensive application storage requirements, Google introduced the **Google File System (GFS)** [38], a large-scale distributed file system that handles hundreds of petabytes in Google's data centers. Its architecture relies on a large number of storage servers that store the 64 MB data chunks into which the files are split. A centralized master server is in charge of managing the file system hierarchy and the locations of the all data chunks. Client operations only include a short interaction with the master server, all data transfers passing through the storage nodes directly. To ensure high-performance data accesses, GFS does not implement POSIX semantics. Instead it employes a weak consistency model, optimizing the file system for access patterns corresponding to many concurrent read accesses on huge files, which are almost never overwritten, but rather appended to. GFS runs on commodity hardware, assuming that component failures are frequent. Consequently, the file system employs chunk replication mechanisms and checksumming to maintain data integrity and reliability.

Hadoop Distributed File System. The Hadoop Distributed File System (HDFS) [95, 100] is a popular open-source file system developed as a part of the Hadoop [13] project. It was designed to run on commodity hardware and to provide petabyte-scale efficient storage for thousands of concurrent data accesses. The architecture of HDFS is similar to GFS, being based on a centralized metadata manager for the file system and a set of storage nodes among which the large files are striped into equally-sized chunks. Furthermore, all I/O operations are performed through the storage nodes, thus allowing the system to achieve the high performance required by data-intensive MapReduce applications devised for the Hadoop platform. Whereas HDFS provides high-throughput simultaneous reads, concurrent write or append operations are not supported. HDFS provides fault tolerance by transparently replicating data at chunk level. It does not implement POSIX semantics, exposing an access interface specifically designed for Hadoop.

3.3 Cloud storage services

Cloud data services can be classified into several categories, according to their purpose and usage. They typically provide users with simple interfaces for uploading and downloading data, offering strong guarantees with respect to data availability and fault tolerance.

3.3.1 IaaS-level services

Amazon Simple Storage Service. Heavily used by various Amazon Web Services, S3 [91] has become the "de-facto" standard for data storage systems at the level of IaaS Clouds. It proposes a simple web interface that gives users access to a highly scalable, reliable

and robust storage service relying on Amazon's infrastructure. S3 was designed with a minimal set of features, enabling an efficient management of a flat, container-based object namespace. It provides containers, denoted *buckets*, which can store an illimited number of *objects*, representing unstructured data. Each object is identified by a unique key and can have at most 5 TB in size. Amazon's S3 implements two REST [28] and SOAP [45] interfaces, exposing operations for creating/deleting buckets and uploading/downloading/deleting objects. Additionally, S3 also includes a *multipart upload* mechanism designed for uploading massive files. It consists in uploading chunks of the file, possibly in parallel to enhance transfer throughput, and then to commit this changes so as to create the new object. Furthermore, S3 introduced object versioning, enabling successive updates to an object to create new versions. S3 does not employ locking mechanisms, providing atomic updates for any specific key and ensuring that only one update completes successfully and all the others fail in the case of concurrent accesses. It builds on an eventual-consistency model for data updates or deletes, in some cases providing read-after-write consistency for the first write operation for a specific object.

Amazon Elastic Block Storage. EBS [7] provides highly-available and reliable block storage volumes that can be associated to Amazon EC2 instances. Once a volume is created, it can be attached to an EC2 instance (only to one instance at a time), on which it appears as a mounted device that can be formatted with a file system. The stored data are preserved beyond the life of the instance. However, to preserve the data after the volume is destroyed, Amazon enables users to create volume snapshots stored into S3. In this way data can be persistently saved and reused by other EC2 instances. EBS comes with a high I/O throughput that outperforms the local disk attacked to EC2 instances.

Cumulus. The Nimbus project recently introduced Cumulus [14], an open-source data storage service that serves mainly as a virtual machine image repository within the Nimbus Cloud. Cumulus is interface-compatible with S3, but benefits from a modular architecture that allows the system to work in conjunction with various storage backends.

Walrus. Walrus [79] is the data storage system shipped with the Eucalyptus IaaS Cloud implementation. It provides persistent storage for objects contained into buckets and identified by unique keys. As Walrus if fully S3-compliant through its REST and SOAP interfaces, it makes user interaction possible through standard tools developed for the Amazon services community. Walrus is designed with the goal of providing efficient storage for virtual machine images in a centralized repository, it can also be used to host user data, similarly to Amazon's S3.

OpenStack Object Storage. The storage service provided by OpenStack [98], is an open-source data service providing object storage similar to S3. It is able to aggregate storage space from multiple servers in a data center to create a persistent storage solution for petabyte-sized data. The OpenStack Object Storage is scalable and reliable, being designed as a completely decentralized storage solution that relies on replication for maintaining data integrity. It provides the same consistency guarantees as Amazon S3, namely eventual consistency for data updates. Additionaly, Swift supports versioning,

high transfer throughput under concurrency and efficient storage for both small and large file sizes. It also scales out when adding new storage nodes, which are automatically configured.

3.3.2 PaaS-level services

Azure BLOB Storage. The BLOB Storage Service [12] was introduced by Microsoft to provide support for storing large unstructured data. It offers a flat storage system where files are saved in containers and can be accessed through a REST-based protocol. Azure provides two types of BLOBS, each of them addressing different requirements. Page BLOBS are designed for efficient random read/write operations on fixed-size BLOBS that cannot grow over 1 TB. Block BLOBS however are optimized for large data storage, implementing a block-streaming and commit mechanism similar to the multipart uploads in Amazon's S3. The BLOB Storage service provides strong consistency guarantees for concurrent access to data.

Azure Table Storage. Microsoft Azure has also proposed a key-value REST-based service for TB-sized structured data, called the Table Storage Service [102]. Data is stored in the form of tables, defined by a *partition key*, which may contain billions of *rows* of data. Each such row is structured in a set of properties. This service is highly scalable, as data is partitioned across a large number of servers. Azure Tables also provides ACID transactions at the level of one table and complex queries, while storing only small amounts of data per row.

3.3.3 Internal Cloud services

Dynamo. Amazon has built Dynamo [24] as a scalable key-value store providing high availability and fault tolerance for a number of its core services. Dynamo cannot be publicly accessed, being used only for storing internal state information for Amazon's services. It targets trusted environments that scale up to a few hundreds of storage nodes and typically hosts small binary objects (less than 1 MB) identified by unique keys. Dynamo implements an eventual consistency model, supporting low-latency concurrent data reads and updates through replication and versioning.

Bigtable. Bigtable [17] is a distributed storage system designed by Google for large-scale structured data. It has been used as a building block in many Google projects, including web indexing, Google Earth, and Google Finance, proving its scalability across thousands of commodity servers. Bigtable exhibits a custom interface that allows users to define dynamic data layouts based on uniquely identified rows. Each row may include a set of various records, but it cannot however be larger than 64 KB. Therefore, even if Bigtable is designed for supporting very large tables, it can accommodate only small records per row. Bigtable ensures high performance data access and high availability, as it relies on Google File System for storage.

3.4 Features and research challenges

In this chapter, we analyzed a series of distributed file systems, emerged to address the needs of scientific applications and Internet services. Parallel file systems aim at providing high-throughput data transfers. Their design targets scalability, which can be achieved by distributing the workload of concurrent clients among a set of storage servers. Such file systems typically provide a POSIX interface, to enable applications to transparently access the distributed entities of the file system without requiring any modification, in the same way they would access a local file system. However, providing POSIX semantics also introduces some important limitations, especially in the case of concurrent updates of overlapping sections of the same file.

This limitation has been addressed by several file systems devised as storage backends in the context of data-intensive computing. Thus, data-processing platforms, such as MapReduce, shifted the storage requirements towards efficient processing of massive datasets and specialized access interfaces. Data-intensive applications came with different access patterns, essentially concurrent fine-grained reads and very few updates. As a result, the file systems that appeared in this context do not need the complex locking-based POSIX interface, offering only dedicated APIs. Furthermore, the file systems specialized for data-intensive workloads target the storage of a limited number of large datasets, in contrast to parallel file systems, which strived to provide distributed metadata management to accommodate many files generated by computations.

Another class of storage systems is represented by Cloud services. The main concern in Cloud environments is providing reliable and easily-accessible services to customers, rather than performance. To cope with these challenges, Cloud storage systems adopt a simple access interface and weak consistency guarantees. Typically, users can access the systems through standard protocols, without being aware of the way data is managed. Open-source Cloud offerings allow administrators to use a traditional file system as a backend for the Cloud service. Thus, they can transparently take advantage of the features exhibited by the underlying storage layer. In this context, the challenge lies in adapting the Cloud services to efficiently support large datasets and fine-grained access to parts of the stored files. An important aspect is interfacing the Cloud service with an appropriate file system, which can both provide reliability and data-availability guarantees, but also high-throughput data transfers and support for highly-concurrent applications.

Chapter 4

Self-* aspects in Clouds

Contents

In the previous chapters we presented the Cloud computing landscape, with a focus on Infrastructure-as-a-Service Clouds. Furthermore, we investigated various data-management solutions for large-scale data-intensive problems and the existing approaches for Cloud data storage. In this chapter we target a more specific problem: we provide a survey of the possible optimization techniques emerged from the area of autonomic computing. We aim at analyzing the self-management mechanisms present in state-of-the-art Cloud frameworks and at identifying their limitations and research challenges.

As the scale, complexity and dynamism of distributed systems is dramatically growing, their configuration and management needs have started to become a limiting factor of their development. This is particularly true in the case of Cloud computing, where the task of managing hundreds or thousands of nodes while delivering highly-reliable services entails an intrinsic complexity. Furthermore, Cloud computing introduces another challenge that impacts on the resource management decisions. In these contexts, self-management mechanisms have to take into account the cost-effectiveness of the adopted decisions.

Autonomic computing has been introduced by IBM in 2001, as a paradigm inspired by the human nervous system [53]. This paradigm aims at building systems that can manage themselves by automatically adapting to the changing environment conditions. An autonomic system is able to continually react to external factors and update its state according to high-level policies. To achieve self-management, a system has to encapsulate a set of self-* properties, including the main four properties defined by the initial IBM autonomic computing

initiative [58, 85, 36, 89]: *self-configuration*, *self-protection*, *self-optimization*, *self-healing*. Subsequent research efforts enhanced this list with new properties conceived to enable or to complement the initial ones, such as *self-awareness*, *self-adjusting*, *self-anticipating*, *self-organizing*, *self-recovery* [97, 96].

We will focus on three of the most relevant features for Cloud storage systems, which help addressing some critical challenges in this context, including security and elasticity:

Self-awareness is the feature that enables a system to be aware of the resource usage and the state of its components and of the infrastructure where they are running. This is mainly achieved through monitoring and interpreting the relevant information generated by the usage of the system. Self-awareness is the key enabling attribute that facilitates the development of any other self-* component, as each of them relies on information collected from the system.

Self-protection addresses the detection of hostile or intrusive actions directed towards the system's components. Self-protecting systems have to identify attacks, protect against unauthorized resource usage by enforcing flexible security policies. Moreover, such a system has to take appropriate measures to prevent detected malicious activities from affecting its functionality and make itself less vulnerable to subsequent similar attacks. Self-protection also includes the capability to handle malicious user identification and to refine user-access policies accordingly.

Self-configuration is the ability to efficiently allocate and use resources, by dynamically adapting the system's deployment scheme as a response to varying and unpredictable environment conditions. The system has to be able to reconfigure on the fly, when its state requires or allows for a change in the number of managed nodes.

4.1 Self-awareness

Large-scale monitoring is an active research area in the context of geographically-distributed environments, ever since the emergence of Grid platforms. In the case of Grids, introspection is often limited to low-level tools for monitoring the physical nodes and the communication interconnect: they typically provide information such as CPU load, network traffic, job status, file transfer status, etc. In general, such low-level monitoring tools focus on gathering and storing monitored data in a scalable and non-intrusive manner [115].

For instance, systems such as Ganglia [65], the Globus Monitoring and Discovery Service (MDS) [70] or the Relational Grid Monitoring Architecture [21] developed within the Open Grid Forum (OGF) [81] are complex hierarchical frameworks designed to provide efficient monitoring services for large clusters and Grid environments. NetLogger [48] or Stardust [107] are systems that target distributed application profiling, with a focus on the identification of performance bottlenecks and predicting the effects of various workloads.

Despite the broad range of existing monitoring systems that focus on generic parameters, little has been done to develop introspection mechanisms specifically targeted at large-scale distributed data management. This is particularly important in the context of data-intensive applications, which require the backend storage system to handle massive amounts of data under heavy concurrency. For such systems, self-awarness involves not only collecting

generic monitoring information, but also analyzing specific parameters. For instance, data distribution, storage space availability, data access patterns or application-level throughput are relevant for data-storage systems.

Nevertheless, data-management systems typically integrate only minimal introspection mechanisms into their design. For example, HDFS implements a heartbeat protocol used by the datanodes to notify the centralized metadata manager about their status. The centralized manager is then in charge of making decisions related to data placement and replication. Similarly, other file systems, such as Lustre or PVFS include such notification mechanisms, but they do not employ in-depth introspection techniques.

In the context of Cloud environments, IaaS Clouds provide tools for the administrator to interrogate the central Cloud controller about the status of the managed nodes. Furthermore, some Clouds also incorporate tools to interface with low-level monitoring systems. For instance, Eucalyptus Clouds implement a set of scripts to configure monitoring systems such as Nagios or Ganglia. At the level of the Cloud data-storage services however, introspection has to be enforced through external monitoring systems, as no mechanisms are built into the studied open-source systems.

4.2 Self-protection

The importance of self-protection in cloud data-management systems has been highlighted by a number of research efforts, dealing with security [56, 1, 44]. We investigated the security mechanisms implemented for some of the reference distributed storage systems.

Distributed file systems. Distributed file systems provide similar approaches to security, essentially based on client access control lists (ACLs). We discuss the example of the Hadoop Distributed File System (HDFS), as it is a file system designed for large clusters and many clients. Furthermore, HDFS is the storage backend for the Amazon MapReduce framework, which provides powerful data processing in Cloud environments. Thus, the file system is exposed to potentially malicious users, as opposed to a typical deployment in local cluster, which can be considered a trusted environment.

In HDFS, security is implemented as a rudimentary file and directory permission mechanism. For authorization, the permission model is similar to that of traditional Linux platforms, each file and directory being associated with an owner and a group. Since both clients and servers need to be authenticated for keeping data secure from unauthorized access, HDFS relies on Kerberos [74] as the underlying authentication system. The main security threats in HDFS arise from the lack of user-to-service authentication, service-to-service authentication and from the lack of encryption for data transfers. Moreover, even if a typical user does not have full access to the file system, HDFS is vulnerable to various attacks that it cannot detect, such as Denial of Service (DoS).

Cloud storage systems. In Amazon's S3, users can decide how, when and to whom the information stored in Amazon Web Services is available. Amazon S3 API provides access control lists (ACLs) for write and delete permissions on both objects and objects containers. Regarding data transfers, data in transit is protected from being intercepted, as the access is

allowed only via SSL encrypted endpoints. Although S3 does not encrypt stored data, uses may encrypt the files before uploading so as to make sure the data are not tampered with.

In Microsoft's Azure cloud platform, access to user data is granted as well by means of authentication and authorization. Several security mechanisms at different layers of the cloud storage infrastructure implement a defense-in-depth approach. The identity and access management ensures that only properly authenticated entities are allowed access. A service-management API provides web services via a REST protocol that runs over SSL, being authenticated with a certificate and private key generated by users.

Open-source systems provide similar mechanisms, limited to ACLs defining user and group access rules. However, no high-level security mechanism is available to protect the environment from complex attacks that cannot be prevented by authentication mechanisms, such as distributed denial of service or continuous data crawling.

4.3 Self-configuration

We investigated the self-configuration mechanisms existing in Cloud environments. More specifically, we focused on a set of data-management systems, as self-configuration is an essential requirement for achieving scalability and elasticity in Cloud contexts.

Parallel file systems. In the case of the parallel file systems, the self-* mechanisms are limited, as they are typically designed for clusters of dedicated machines, which do not usually change their configuration. However, some file systems include self-management aspects to improve their data availability and reliability.

We take GPFS as a case study, since IBM enhanced it with a set of automatic configuration features. First, it provides an information lifecycle management component that allows administrators to control data placement through high-level policies. The policy engine can create storage pools appropriately tuned for specific applications and to transparently configure and manage several storage tiers. Furthermore, a GPFS cluster supports a dynamic addition or removal of storage servers, being also designed to rebalance data placement upon modifying the set of storage nodes. However, this feature targets manual storage pool extension or node removal in case of failures. For instance, it does not address an automatic decrease of the number of used storage nodes when many servers are idle as a result of low application load. Additionally, GPFS does not include a configuration component targeted at estimating and updating the size of its storage pools, relying only on user defined settings.

Dedicated file systems. We consider the two distributed file systems designed for MapReduce data-intensive workloads, the Google file system and HDFS. Such file systems rely on commodity hardware for their storage servers, thus being built with the assumption that component failures are the norm rather than the exception.

Therefore, they include mechanisms for constant monitoring of the active storage nodes and automatic data recovery and replication. GFS implements automatic replica distribution mechanisms that takes into account the location of the data storage servers. Moreover, replicas are examined periodically and redistributed to other storage servers to improve disk

space utilization and load balancing. In contrast, HDFS has only limited support for data re-replication and load balancing among storage nodes. Both systems support the addition of new storage servers, which are gradually filled with existing chunks to avoid redirecting all writes to the new servers and thus create bottlenecks.

However, both systems lack complex self-adaptation mechanisms that require comprehensive information related to data distribution and access patterns to optimize the system behavior on the fly.

Cloud storage. In the area of Cloud storage, we can identify several approaches to self-management, which vary according to the targeted usage of the storage systems. The implementation and internal architecture of commercial Cloud storage systems is usually proprietary, as it is the case for Amazon or Azure storage offerings. On the other hand, the storage system proposed by OpenStack is a storage solution very similar to S3 with respect to the targeted workload. Thus, it is built to host large datasets in a distributed and reliable fashion. OpenStack Storage implements some limited self-management mechanisms, mostly dealing with maintaining data replication upon failures. Furthermore, as it targets commodity hardware, the system also integrates mechanisms to handle the node arrival into or removal from the storage servers pool.

Nevertheless, most open-source Cloud storage offerings do not target heavy workloads, being designed primarily for storinh virtual machine images. They typically lack self-configuration mechanisms, relying on the backend storage layer to handle failures or data replication.

4.4 Summary

In this chapter we introduced the *autonomic computing* paradigm and we investigated the potential benefits of introducing self-management mechanisms at the level of data-management services, in particular in Cloud environments. We considered three essential self-* aspects and we analyzed the existing autonomic mechanisms already implemented in various data-management systems. *Self-awareness* provides the knowledge needed by higher-level self-management components to make decisions. *Self-protection* plays a crucial role in untrusted environments like Clouds. Finally, *self-configuration* enables systems to dynamically adapt to changing external conditions.

Chapter 5

Case Study: BlobSeer, a versioning-based data management system

Contents

This chapter introduces BlobSeer [77, 76], a concurrency-optimized data-management system for data-intensive distributed applications. In the previous chapters, we discussed the features and limitations of existing data-storage solutions for data-intensive applications, as well as of the data-management approaches emerged at the level of IaaS Clouds. BlobSeer specifically targets applications that handle massive unstructured data in the context of large-scale distributed environments. We selected BlobSeer as a case study throughout this thesis, as it can be used to overcome some of these limitations and to provide new perspectives in the context of Cloud data storage.

5.1 Design principles

To meet the specific requirements of data-intensive applications for large- scale infrastructures, the BlobSeer system was designed to comply with the following principles:

Massive unstructured data. In BlobSeer, data is organized as a set of large, unstructured sequences of bytes, denoted BLOBS (Binary Large Objects). Each BLOB is uniquely identified in the system by a globally shared identifier. This approach presents two advantages. First, it ensures *data-location transparency*, therefore allowing users to access data only by knowing a simple identifier, instead of being aware of the location of a specific piece of data. Furthermore, as BlobSeer addresses the management of huge datasets that can easily go beyond TB sizes, it has to be inherently *scalable* to efficiently store and provide access to such data. To this end, compacting many KB-sized files generated by distributed applications into huge files enhances scalability by reducing the management overhead of many filenames and associated namespace hierarchies. However, to provide a useful tool for processing such data, BlobSeer also provides *fine-grained access* to the stored data sequences, enabling concurrent processes to retrieve the needed data blocks without needing to sequentially search through the whole BLOB.

Data striping. Many distributed file systems employ data-striping techniques to improve the performance of data accesses. In BlobSeer, each BLOB is split into equally-sized *chunks* which are distributed across multiple storage nodes. The size of each BLOB is specified by the user, so that it can be fine-tuned according to the needs of the applications. For instance, we can consider a BLOB that is fragmented into large chunks and an application that needs to access a specific sequence of bytes belonging to a single chunk. In this case, concurrent processes that read the same data chunk may create bottlenecks at the level of the storage server that hosts the chunk. In contrast, if the data chunks are too small, the overhead of identifying and transferring multiple chunks to a single computing process may prevail over the benefits of striping the data. Hence the need to carefully adapt the size of the data chunks according to the requirements of the application that processes them. Data striping is a popular technique because is also enables load balancing in the context of concurrent reader or writer processes. BlobSeer is able to achieve high aggregate transfer rates due to the balanced chunk distribution among storage nodes, especially when considering many simultaneous clients that require access to non-overlapping chunks of the same BLOB.

Distributed metadata management. In order to provide scalability at very large scales, along with a low overhead for data accesses, several file systems devised for petabyte- sized data adopted distributed metadata management schemes. In BlobSeer *metadata* denotes the information needed to map the location of each BLOB's chunks on the storage nodes. Each chunk is uniquely identified in the system by its BLOB *identifier*, *offset* within the BLOB and *size*. Such information is stored on specifically designed nodes and is employed by the users to discover the location of each chunk that has to be retrieved. Distributing metadata also has an additional advantage, namely it can eliminate *single points of failure* when the metadata are replicated across multiple servers.

High throughput under heavy concurrency. This prerequisite is implemented in BlobSeer through a *versioning-based* concurrency control. In BlobSser, data is never overwritten. Instead, each new WRITE performed on a specific BLOB results in a new version. Each BLOB version stores only the differential update with respect to the previous versions, but exposes the whole BLOB obtained as a result of the WRITE operation. This approach enables an effi-

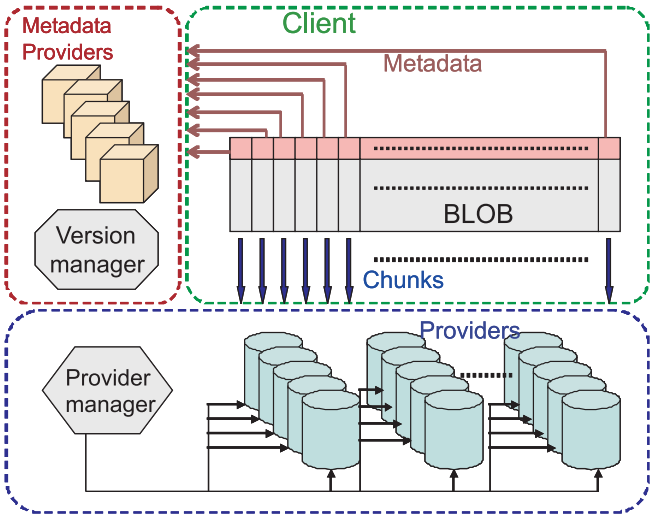


Figure 5.1: The architecture of the BlobSeer system.

cient implementation of concurrent updates of the same BLOB, by considering all data and metadata immutable. Thus, concurrent writers can create write their updates into the system, which in turn is responsible for serializing the writes and assigning a consistent version to each of them after all data has been successfully sent to the storage nodes. Furthermore, a versioning-based design allows for a complete decoupling between concurrent READ and WRITE operations performed on the same BLOB. While a writer is in the process of creating a new BLOB version, multiple concurrent readers can safely access the previous versions, as the new updates never modify the data already stored into BlobSeer.

5.2 Architecture

The architecture of BlobSeer is based on a set of distributed entities illustrated on Figure 5.1.

Data providers. To provide a scalable and efficient storage service, BlobSeer relies on multiple *data providers* to host data chunks. Each *data provider* is implemented as a high-performance key-value store, which supports fast upload or download of data chunks. This in-memory cache is backed by a persistency layer built on top of BerkleyDB [114], an efficient embedded database. Furthermore, the *data providers* rely on an RPC layer, detailed in [75], which consists in an asynchronous communication layer employed by all BlobSeer entities.

Provider manager. The *provider manager* is responsible for assigning *data providers* to the WRITE requests issued by users. It keeps track of all the *data providers* in the system. To select a *data provider* to store a new data chunk, the *provider manager* implements a configurable strategy. Each *data provider* periodically sends informations about its state to the *provider manager*, which is used to compute a store reflecting the *providers* capacity to host new data. The default strategy is to select the provider that stores the smallest number of chunks and the smallest number of pending WRITE requests. More complex

strategies can be implemented is the system has access to more detailed information about the state of the *data providers* and the user requirements.

Metadata providers. To keep track of the chunks distribution across *data providers*, each BLOB is associated with a set of metadata. For each BLOB, the metadata is organized as a *distributed segment tree* [117], where each node corresponds to a version and to a chunk range within that version. Each tree node stores the location of its two descendants, each of them being responsible for half of the parent's chunk range. Each leaf covers just one chunk, recording the information about the data provider where the page is physically stored. The metadata trees are stored on the *metadata providers*, which are processes organized as a Distributed Hash Table [9, 22]. Their implementation is similar to that of the *data providers*, metadata tree nodes being stored in a key-value cache, backed by a database for persistency.

Version manager. The *version manager* deals with the serialization of the concurrent WRITE requests and with the assignment of version numbers for each new WRITE operation. Its goal is to create the illusion of instant version generation, so that this step does not become a bottleneck when a large number of clients concurrently update a specific BLOB.

Client library. BlobSeer provides a *client* library to make available its access interface to higher-level applications. The *client* supports the following operations: CREATE BLOBS, READ, WRITE or APPEND contiguous ranges of bytes. The BlobSeer system is designed to handle many concurrent client operations accessing the same BLOB or different BLOBS.

BlobSeer User. We define a BlobSeer *user* as any higher-level application which employs the *client library* to access the BlobSeer system. A user may coordinate multiple concurrent *clients* on a single machine or on a distributed set of physical machines. As there are no authentication mechanisms implemented in the current version of BlobSeer, we assume a user only utilizes a single physical node and can be identified by its IP address.

A typical setting of the BlobSeer system involves the deployment of a few hundreds data providers, storing BLOBS of the order of the terrabytes. The typical size for a chunk within a BLOB can be smaller that 1 MB, whence the challenge of dealing with hundreds of thousands of chunks belonging to just one BLOB. BlobSeer provides efficient support for heavily-concurrent accesses to the stored data, reaching a throughput of 6.7 GB/s aggregated bandwidth for a configuration with 60 metadata providers, 90 data providers and 360 concurrent writers, as shown in [76].

5.3 Zoom on data access operations

The CREATE operation.

The CREATE primitive exposed by the BlobSeer client library only involves a request for the *version manager*. The user has to specify the *chunk size* required for the new BLOB and the *replication degree* needed for each of its chunks. In BlobSeer, each data chunk is replicated on

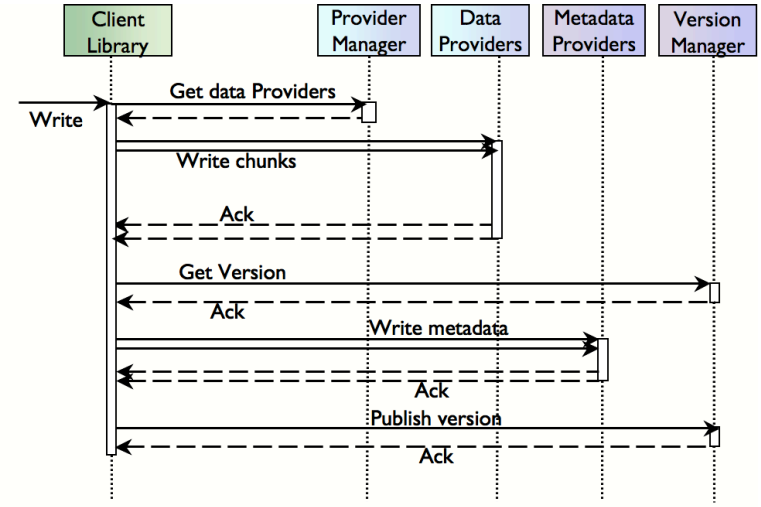


Figure 5.2: Sequence diagram of the WRITE operation in BlobSeer.

several *data providers*. The number of replicas, denoted *replication degree*, has to be specified when the BLOB is created and therefore is constant for all data chunks belonging to the same BLOB. Nevertheless, the replication degree can vary across BLOBS, according to the application needs in terms of data reliability.

The WRITE/APPEND operation.

The WRITE operation implemented by the BlobSeer client, depicted in the diagram on Figure 5.2, consists of two main steps, sequentially executed by the client:

Data-writing step. This is the first operation executed when a user calls the WRITE primitive for a specific BLOB and a contiguous range of chunks delimited by the *offset* of the first chunk to be written and the *size* of the entire sequence. To write such chunk ranges on the *data providers*, the client library performs the following steps:

- Contact the *provider manager* and ask for a number of *data providers* equal to the number of chunks that need to be written, multiplied by the number of replicas for each chunk.
- Upload the data chunks to the received *data providers*, in parallel. The chunks are sent in an asynchronous fashion, allowing all the uploads to make progress in parallel and thus to achieve a high transfer throughput. This step is successfully completed when at least one replica of each chunk has been correctly uploaded onto the corresponding *data provider*.

Data-publication step. To make the uploaded data chunks available to the users as a new BLOB version, the WRITE operation includes a second phase. This phase is detailed below:

- The client contacts the *version manager* to notify it about its WRITE operation. The *version manager* assigns the client a version number and adds the WRITE into a queue containing in-progress WRITES.

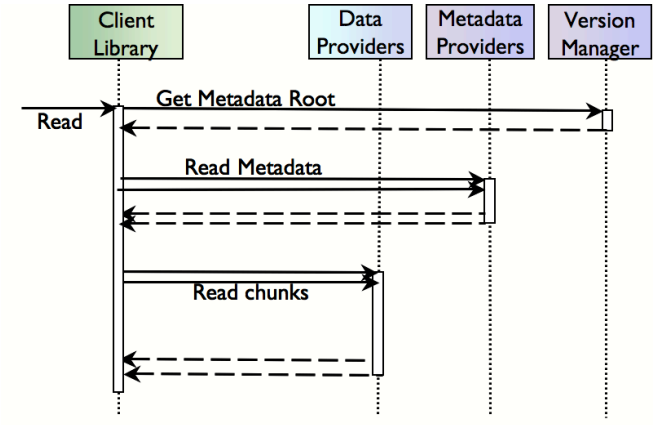


Figure 5.3: Sequence diagram of the READ operation in BlobSeer.

- The client constructs a metadata tree associated with the new version, so that the leaves corresponding to the written chunk range store the location of the *data providers*. Only the leaves for the new chunks are built. To create a complete tree that reflects the latest version of the whole BLOB, the metadata tree is weaved against the previous versions. Once the metadata nodes are created, the client sends them in parallel to the *metadata providers*.

- Finally, the *version manager* is notified that the metadata associated with the new version are ready. At this point, the client library has successfully completed the WRITE operation. The *version manager* is in charge of serializing the possible concurrent WRITES to the same BLOB and to publish the new version.

APPEND operations are similar to WRITES, the only difference being that the client does not have to specify the *offset* where it initiates the operation. The *version manager* takes care of assigning an *offset* according to the size of the latest version recorded for the BLOB.

The READ operation.

BlobSeer is designed to support high-throughput READ operations, in particular in the context of multiple concurrent clients accessing the same BLOB. This is achieved through a set of operations implemented at the level of the client library, which leverages the distributed design of both data and metadata, as shown on Figure 5.3:

- In a first step, the client has to retrieve the root of the metadata tree corresponding to the specific BLOB identifier and BLOB version requested in the READ call. To this end, the client contacts the *version manager*, which stores the root information for all the BLOBS in the system.

- Next, the client scans the metadata tree by issuing metadata read requests on the *metadata providers*. It only traverses a portion of the tree, the one that covers the needed chunk range. By reaching the leaves, the client can retrieve the location of the *data providers* that physically store the data chunks.

- The client downloads the data chunks in parallel from the emphmetadata providers. Being able to asynchronously retrieve multiple chunks in the same time, the client completes the operation in an efficient manner. If there is at least one chunk for which no replica can be downloaded, the client returns an error code, as the whole READ operation is considered to have failed.

5.4 Summary

This chapter details the architecture and data-management techniques implemented in BlobSeer, a data-management system for data-intensive distributed applications. We target data-storage platforms designed to manage massive unstructured data distributes across large-scale environments. As BlobSeer inherently addresses such requirements, we rely on BlobSeer as a case study to validate various self-management mechanisms in the following chapters. Furthermore, we investigate the possible integration of BlobSeer as a Cloud storage service. The main prerequisites of such a system match the design principles on which BlobSeer was built: scalable architecture, efficient handling of massive data, high throughput for data accesses and optimizations for concurrent data transfers.

Part II

Enabling BlobSeer with self-management

Chapter 6

Self-management for distributed data-storage systems

Contents

This chapter presents our contribution with respect to self-management in the context of large-scale, distributed storage systems. In Chapter 3, we introduced several existing data-storage approaches and we presented their specific limitations, focusing on a series of relevant self-* research directions for improving the management of massive data. In this chapter, we propose a set of generic components designed to overcome these limitations by enhancing storage systems with self-* properties.

Our goal is to improve the performance and the efficiency of the resource usage in a data-storage system by enabling an autonomic behavior tailored to its specific requirements, such

as changing rates of concurrent users, management of huge data spread across hundreds of nodes or malicious attempts to access or to tamper with stored data. To this end, we designed three self-management frameworks described in the following sections.

6.1 Self-awareness: Introspection mechanisms

In the area of distributed systems, introspection mechanisms play a crucial role in assisting the users in overcoming the challenges raised by the behavior of their systems at large scales. Introspection typically relies on monitoring tools, which provide the users with the feedback necessary for identifying the state of their application and the state of the infrastructure where the application is running on, at a particular moment in time.

This section discusses the relevant monitoring information that can serve as an input for any self-adaptation engine. It then proposes a generic architecture to enable self-awareness capabilities for a data management system, as this is the first building block towards any other self-* component.

6.1.1 Relevant data for storage systems

Introspection is the prerequisite of an autonomic behavior, the first step towards improving the performance and optimizing the resource usage of data-storage systems distributed at large scales. In such a context, we identified a set of parameters that need to be monitored and analyzed, including general information about the running nodes, data distribution, storage space availability, data access patterns, application-level throughput, etc. These parameters can be classified as follows:

General information. Such data is essentially concerned with the physical resources of the nodes that act as storage providers. They include CPU usage, network traffic, disk usage, storage space or memory.

A self-adapting system has to take into account information about the values of these parameters across the nodes that make up the system, as well as about the state of the entire system. For instance, the used and available storage space at each single storage node play a crucial role in deciding whether additional nodes are needed or not. Besides the basic information related to each node, the system also needs access to aggregated data, such as the value of the total storage space occupied/available for the entire system.

System events. The most significant information for a single file is its access pattern, i.e., the way the recorded data are accessed through READ and WRITE operations. As an example, in the case of a data-management system based on data striping, a basic monitoring parameter is the number of READ or WRITE accesses for each data chunk. Since each WRITE or READ operation consists in accessing a range of consecutive chunks, it is expected that some ranges of chunks will have the same number of accesses. As a consequence, such data facilitate the identification of chunk ranges with a high rate of accesses within the file. This can prove to be a valuable information for the replication algorithms, which can assign more replicas to the chunks that are highly accessed by the clients.

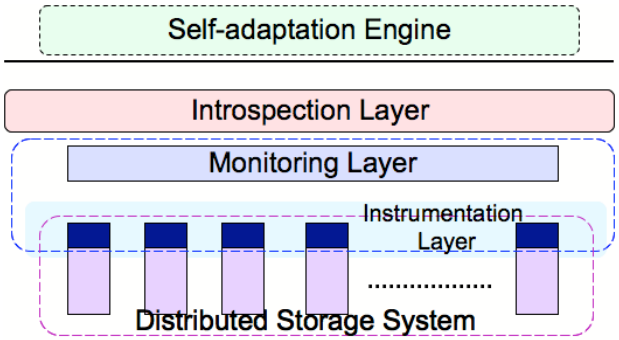


Figure 6.1: Global architecture of a self-awareness framework.

Finally, the yielded data may characterize the history of the user accesses to data. This information can be used to detect users that attempt to damage the stored data or to attack the system by misusing the access primitives provided by the interface. An example of an attack that can be blocked by analyzing this type of information is Denial of Service, in the form of attempts to overload the storage nodes and turn them into unresponsive ones.

Global state. Even though the self-adaptation engine has access to the details associated with each event, an efficient decision-making component may require a higher-level overview of the state of the system and the stored data. As the system has to accommodate huge amounts of data and a large number of concurrent clients, keeping track of each event in the system is a time-consuming task. As a consequence, to enable a rapid evaluation of the system state, the self-awareness component should process the collected parameters and yield system-wide aggregated information.

An example of some key aggregated data is the total number of accesses associated with each storage node. This is a measure of the load of each of them and can directly influence the selection of the nodes that will be used to store new data. Another system-wide data refers to the distribution of the recorded data among storage nodes. To take the example of a system that employs data striping, it is important for the node allocation strategies to be aware of the way data is managed. Taking into account such information, namely the sizes of the data slices that are hosted on each node, can help improve load balancing among data nodes. It can be equally useful to expose the files that have a high rate of change or growth, as opposed to the ones that contain chunks that are seldom modified. The dynamic growth of a file can be characterized by the number of performed WRITE operations, as well as by the modification rate of its size.

6.1.2 Global architecture

In order to enable a data-storage platform with introspection capabilities we have designed a three-layered architecture aiming at identifying and generating relevant information related to the state and the behavior of the system. The global architecture, presented in Figure 6.1, consists of independent layers that fulfill specific tasks:

Introspection Layer. Its role is to process raw monitoring parameters received from the *Monitoring Layer* and to store them in a persistent fashion. Thus, the *Introspection Layer*

can provide an accurate image of the state of the system and generate relevant input data for higher-level self-adaptation engines. The data yielded by this layer include aggregated information specifically structured to suit the component that further needs it: as an example, a self-protection component will require monitoring information oriented towards the client actions, such as statistics about the number and types of data accesses per client.

Monitoring Layer. This layer is responsible for collecting the data from the lower *Instrumentation Layer* and for relaying it to the upper layers in a reliable and efficient way. Moreover, it has to be able to handle user-defined monitoring events and to accommodate huge amounts of monitoring data generated when multiple users simultaneously access the system.

Instrumentation Layer. It enables the storage system to send monitoring data to the upper layers. It can be implemented in two ways: on the one hand, it can be represented by blocks of instrumentation code injected into the source code of the monitored system; this method assumes that the administrator has access to the source code of the system. On the other hand, the instrumentation layer can access a set of logs generated by the system and forward the information to the monitoring layer. Although simpler, this solution may slow down the reaction of any self-* component built on top of the introspective framework.

6.2 Self-protection: A generic security framework

In this section we propose a generic security management framework to enable self-protection for Cloud data-management systems, by allowing service providers to define and enforce complex security policies. We focus on a series of essential requirements:

Reactivity. To achieve self-protection, the system should enforce a reactive loop; that is, after detecting a malicious attack and if required by the security policy in place, take appropriate actions against the client who initiated the attack and bring the system back into a consistent state.

Non-intrusiveness. The security mechanisms have to be designed such that they do not impact data-access performance, allowing the clients to fully utilize the resources of the targeted data management system.

Portability. The system should be easily interfaced with various storage systems.

Flexibility. To be effective against a wide range of attacks, the security framework must be able to handle and enforce customized security policies.

With these challenges as a starting point, we first investigate a set of scenarios that justify the need for security mechanisms specifically designed for data-management systems. In contrast to other distributed platforms, such as clusters or Grids, Cloud environments are vulnerable to malicious attacks. Thus, we focus on security issues specific to Cloud storage systems. Next, we introduce our security framework and we discuss the design details of its components. Finally, we focus on the policy definition language and the algorithms employed to detect policy violations.

6.2.1 Motivating scenarios

The following scenarios illustrate some representative applications for a Cloud storage platform and examine the inherent security threats of their usage patterns. These motivating scenarios highlight the benefits of complementing conventional authentication and authorization mechanisms for Cloud data services with a security management framework. Its goal is to enable service providers to supervise user actions and to restrict activities that fall outside the normal usage.

Cloud storage for video surveillance: Video surveillance cameras typically generate a continuous data flow that requires a large amount of storage space. The data will not be written to a single file, as video surveillance cameras usually store the recordings to different files according to their timestamps. A suitable storage system has to be able to scale to a large number of cameras, each of them concurrently writing huge amounts of data to different files. To address such needs for storage capacity, the data can be hosted directly in the Cloud. In this scenario, an attacker might attempt a DoS attack on some of the storage nodes by sending a large number of write requests. This would lower the response time of the attacked data storage nodes, thus affecting the rate at which the data can be stored for the entire system. In order to maintain the overall performance at an acceptable level, these attacks must be quickly identified and blocked.

Storing medical records in the Cloud: In this scenario we consider a medical center which stores all the medical records for its patients in the Cloud. The employees have access to all the files, but each of them is supposed to access only the documents related to his work. The main security concern in this case is that we must protect the data from being accessed by unauthorized users. An attacker can impersonate an authorized user by stealing its credentials, and then attempt to read all the stored files (crawling). This kind of unexpected behavior (reading all records in a short period of time) has to be detected as being suspect, since it can expose a compromised user. However, this is not a clear indication of an attack since an authorized user may as well perform this kind of actions. As a result, such behavior has to be labeled as suspicious. Yet, it will not result in a penalty for the client until it is correlated with other detected attacks.

Such threat scenarios represent complex attacks against which typical authentication and authorization mechanisms are vulnerable. To be able to identify such malicious intrusions, we have designed a flexible and extensible language to describe the access patterns specific for each type of attack. Moreover, we have developed a security management framework to detect and possibly block any client attempting an attack described by such patterns.

6.2.2 Global Architecture

In order to provide a high-level security mechanism for Cloud storage systems, we propose a generic framework for both security policies definition and enforcement. Figure 6.2 illustrates the modular architecture of our framework and the interactions between the components.

The Policy Management module represents the core of the framework, where security policies definition and enforcement takes place. This module is completely independent

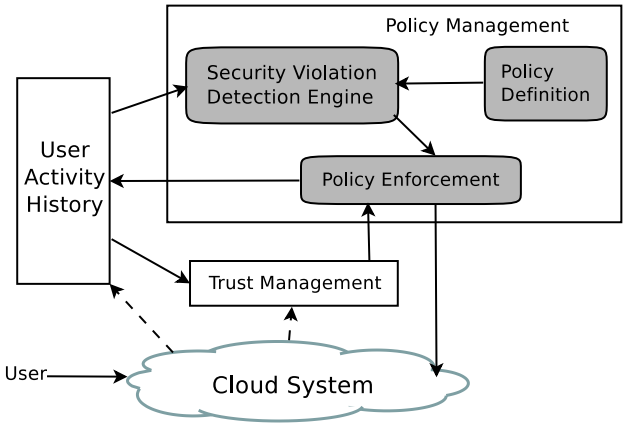


Figure 6.2: High-level architecture of the security management framework.

of the Cloud system, as its input only consists in user activity events monitored from the system.

The User Activity History module is a container for monitoring information describing users' actions. It collects data by employing monitoring mechanisms specific to each storage system and makes them available for the *Policy Management* module.

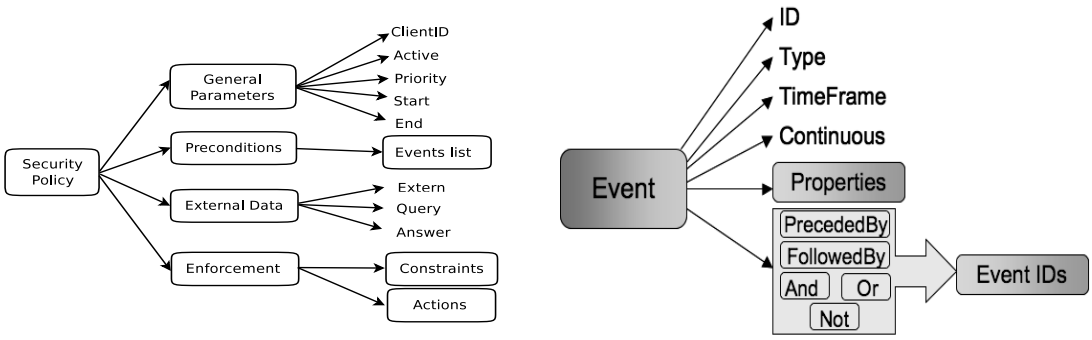
The Trust Management module incorporates data about the state of the Cloud system and provides a *trust level* for each user based on his past actions. Identifying users as fair or malicious, the *Trust Management* module enables the system to take custom actions for each detected policy violation.

We focused on the *Policy Management* core. In order to have an adequate malicious client detection level, we first have to define what kind of behavior is considered inappropriate or dangerous for the system. This is done through the *Policy Definition* component, which provides a generic and easily extensible framework for defining various types of security policies. The *Security Violation Detection Engine* scans the *User Activity History* in order to find the malicious behavior patterns defined by the security policies. When such an attack is detected, the *Policy Enforcement* component is notified and a set of possible feedback actions are forwarded to it. The *Policy Enforcement* component is responsible for making a decision based on the state of the system and on the impact of the attempted attack on the typical performance of the system. Such decisions range from preventing the user from further accessing the system to logging the illegal usage into the *User Activity History* and decreasing the *trust level* corresponding to that user.

6.2.3 Security policies

In this section we show how we define templates for various attacks and how we map them into security policies. Then, we give an insight on the mechanisms we designed to detect such attacks.

In order to detect the various types of attacks that the user actions can expose, our policy management module has to meet a set of requirements:



(a) High-level representation of a security policy.

(b) Structure of an event.

Figure 6.3: Defining security policies.

Generic format. The format of the security policies must be independent of the type of data collected by the *User Activity History* module, so as to enable the *Policy Management* module to dynamically interface with various systems.

Flexibility. The format used to describe the security policies has to be flexible and expressive enough to allow the system administrator to translate any type of attack into a policy that can be understood by the *Policy Management* module.

Extensibility. This is an essential feature of the security policies, as specific attacks need an enriched policy format according to particular events collected by the user activity history.

Simple API. Complex policies may include a wide set of elements linked together by logical and time dependencies. Defining such policies and all the appropriate parameters for each of their components can be a tedious and error-prone task. This process should thus be automated by means of an API that allows an intuitive definition of security policies compliant with our format.

We defined a hierarchical format for the security policies, so as to comply with the above requirements. On the one hand, each policy contains a set of template user actions that make up a pattern corresponding to a particular security attack. In addition, the policy can specify a set of thresholds that draw the limits between normal behaviors that exhibit the same activity pattern and malicious user actions.

In order for an attack to be detected, the policy has to be instantiated for a specific user, that is, the activity history of that user has to include recorded actions that match the template sequence provided by the policy. As an example, a DoS attack can be defined by a series of write operations that take place in a short period of time and are initiated by the same client. Therefore, the corresponding policy will describe a write operation as the needed pattern and will specify a duration and the maximum number of write operations considered normal for that duration.

On the other hand, a security policy has to specify a set of actions that are forwarded to the *Policy Enforcement* module when the policy is instantiated and thus a malicious user is identified. These actions include feedback specific for the Cloud system, and recording the policy violation into the *User Activity History*.

Figure 6.3(a) illustrates the tree structure of a security policy. It consists of four elements:

The template set of user actions. The *Preconditions* element encloses the sequence of user actions that describe the pattern of an attack. Each user action is modeled by an *Event*, described through a set of attributes that identify a particular type of records in the *User Activity History*. To take the example of the DoS attack again, the *Preconditions* may contain only one event, whose *Type* attribute points to the list of recorded write operations in the *User Activity History*.

General Parameters. They are used to differentiate the policies (e.g., *Active*, *Priority*) and to enable the detection module to interpret the events describing the policy by specifying the *Start* and the *End* event.

Actions suggested when the policy is instantiated. The element *Enforcement* contains a list of *Constraints* and *Actions*. When the sequence of events defined by the policy is matched, the *Security Violation Detection* module will select the satisfied *Constraints* and propose the associated *Actions* to the *Policy Enforcement* module, which will be in charge of executing them. This approach allows us to define flexible policies that result in a customized feedback that depends on some given constraints.

Interaction with external modules. The element *External Data* allows a policy to receive auxiliary input data from external modules, in addition to the *User Activity History*. For instance, a policy may need the user's access control list (ACL) to make a decision, but this data has to be queried from an external ACL module and is not present in the *User Activity History*. This element enhances the extensibility of the policy format, allowing administrators to plug specific system building blocks to the *Policy management* module.

Figure 6.3(b) shows the structure of an *Event*. It includes a *TimeFrame* element that allows for the event's positioning in time with respect to one or more events in the same policy. To this end, the event also includes *PrecededBy* or *FollowedBy* elements, which enclose references to other events *ID* field. In order to have a more flexible policy definition language, the referenced events can be grouped by means of logical operations such as *AND*, *OR* or *NOT*. The structure of an *Event* further contains an element that models a sequence of user actions that have the same type; for instance, the *Continuous* element is used when modeling DoS attacks, for which the detection module has to look for a large number of similar write operations. Aside from these basic elements, each event can be enriched with attributes containing specific information recorded in the *User Activity History*. Each such attribute can have associated thresholds that allow the detection engine to filter the events that do not match the purpose of the policy. These attributes are grouped in an additional *Properties* element. Taking the example of the DoS attack again, the *Properties* element of the write event should include the number of write operations collected by the *User Activity History* and the size of the written data.

6.2.4 The policy breach detection algorithm

The detection engine is able to handle any type of policy described using the above format, regardless of their complexity or targeted attacks. Its main goal is to search for recorded user

Algorithm 1 Policy Matching Algorithm

```
1: P_partiallyMatched ← ∅
2: procedure PolicyMatch
3:     for p ∈ P ∪ P_partiallyMatched do
4:         startEvent ← p.getStartEvent()
5:         for all c ∈ ClientIDs in parallel do
6:             matchStatus ← p.getMatched(c)
7:             MatchEvent(p, c, startEvent, matchStatus)
8:         end for
9:     end for
10: end procedure
```

actions that match the template events defined by the policy. The attributes are specific to each type of event and they allow the detection engine to identify the required user actions within the activity history.

In the previous section we discussed the structure of the template security policies that can be defined by system administrators. We introduce the notion of *partially matched policy* as a policy for which the *Security Violation Detection Engine* has found monitored user events in the *User Activity History*, so that they match some of the template events in the policy.

The detection algorithm receives a list of policies as input, each of them having a specific priority. The algorithm attempts to periodically detect attacks, according to the priority of each policy. For each template or partially matched policy, it builds a query to the *User Activity History*, attempting to instantiate the next template event in the policy's *Preconditions*. It adds to the list of partially instantiated policies all the possibilities for continuing the match, according to the query's results. The detection process is complete when all the events in a policy are instantiated, that is the history of the user actions reflects a chain of events that are specific to the security attack described by the matched policy.

The policy-violation detection mechanism is presented in Algorithm 1. The *PolicyMatch* procedure is periodically repeated for each type of policy, according to their priority. The list of initial template policies is denoted P and the list of partially matched policies is denoted $P_{partiallyMatched}$. Each template policy is evaluated for each client registered in the system and each partially matched policy is evaluated for the client for which it was instantiated. The *MatchEvent* procedure in Algorithm 2 is the core of the attack-detection mechanism, as it is in charge of matching the template sequence of events in the policy against the data recorded into the *User Activity History*. The detection mechanism implemented by Algorithm 2 is detailed below:

- Initially, the algorithm is executed for the start event specified by the policy. Subsequently, it will recursively try to match each event in the dependency tree described by the policy through the *FollowedBy*, *PreceededBy* and logical *And*, *Or*, *Not* event attributes.

- For each event, the *PolicyMatch* procedure identifies the specific attributes in the template *Properties* and the time constraints associated with the event and tries to fill them in with real data from the *User Activity History*.

Algorithm 2 Event Matching Procedure

```
1:  procedure MatchEvent(p, clientID, event, matchStatus)
2:      eventType ← event.getType()
3:      eventProp ← userHistory.getPropertiesList(client, eventType)
4:      constraints ← event.getConstraints(eventProp)
5:      timeFrame ← event.getTimeConstraints()
6:      foundMatch ← False
7:      repeat
8:          eventData ← userHistory.query(eventProp, constraints, timeFrame)
9:          if eventData = ∅ then
10:             if timeFrame.isValid() then
11:                 Wait event.getWaitTime()
12:             else
13:                 return ∅
14:             end if
15:         else
16:             foundMatch ←True
17:         end if
18:     until foundMatch
19:     p.fillEventPropValues(event, eventData)
20:     p.updateEventTimeConstraints(event, eventData)
21:     nextEvents ← p.getEventDependencies(event)
22:     for ev ∈ nextEvents do
23:         ev.fillParentConstraints(event)
24:         newMatch ← MatchEvent(p, client, ev, matchStatus)
25:         matched ← newMatch.checkLogicalDep(event)
26:         matchStatus ← matchStatus ∪ matched
27:     end for
28:     if event = p.getStartEvent() then
29:         if matchStatus ≠ ∅ then
30:             P_partiallyMatched ← P_partiallyMatched ∪ {p}
31:         end if
32:     end if
33:     return matched
34: end procedure
```

Algorithm 3 Policy Enforcement Algorithm

```
1:  procedure PolicyCheck
2:      for p ∈ P_partiallyMatched do
3:          completeMatch ← p.checkAllEvents()
4:          if completeMatch then
5:              constraints ← p.getActionConstraints()
6:              for all c ∈ constraints do
7:                  if c.isSatisfied(p) then
8:                      a ← p.getAction(c)
9:                      invoke ExecuteAction(a) on PolicyEnforcementModule
10:                 end if
11:             end for
12:         end if
13:     end for
14: end procedure
```

- If the template cannot be matched against the recorded data, the algorithm proceeds into a waiting state. That is, the execution is suspended for an amount of time specified in the policy definition and retrieved through the *getWaitTime* procedure. When the waiting time expires, the algorithm moves on to the next step if the *Activity History* receives relevant data. Otherwise, it repeats the current step until the event's *Time Frame* constraints can no longer be satisfied by new records in the *Activity History*, in which case the policy is discarded.

- If matching data have been retrieved from the *User Activity History*, the event's *Properties* and *TimeFrame* are filled in through the *fillEventPropValues* and *updateEventTimeConstraints* procedures.

- Then, the algorithm calls the *getEventDependencies* procedure to identify he next template events to be matched, that is the events that are linked to the current event through dependency attributes. Each such event's constraints are filled in with preliminary values derived from the properties of the parent event.

- Next, the procedure is recursively called for the new events until it fills in the whole set of template events in the policy or it cannot find any matching data in the user history.

- After the first event and its dependency tree are completely processed, the policy can be added to the set of partially matched policies. They will in turn be fed to both the *PolicyCheck* and the *PolicyMatch* algorithms until each policy is either successfully completed (i.e. it detects an attack) or it cannot be filled in with real data for a specific client and gets discarded.

Algorithm 3 scans all the partially matched policies. The attack described by such a policy is detected for a specific client when all the template events for the policy are filled in with real data. In this case, the *PolicyCheck* procedure will inspect the actions associated with the policy and trigger the *Policy Enforcement* module for the action that corresponds to the constraints defined by the policy.

6.3 Self-configuration: Dynamic dimensioning

A means to achieve self-configuration in a data storage system is to enable the number of storage nodes to scale up and down depending on the detected system needs. As an example, let us consider a IaaS Cloud environment, such as the Amazon EC2, where the user is running a distributed storage system on top of a set of Amazon instances. When the system is overloaded, the user may need to expand the pool of used machines by renting new nodes from the Cloud provider.

To this end, Amazon introduced the *Auto Scaling* [6] service that can automatically scale up and down the number of running instances belonging to a user. However, the *Auto Scaling* service makes decisions based on simple monitoring data (such as the CPU load of the nodes) and cannot take into account more complex factors that can impact on the performance of a specific system.

To adapt the storage system's behavior to the changing state of the environment, we designed a component able to contract and expand the pool of storage nodes based on configurable parameters monitored from the storage system, such as the rate of data accesses.

6.3.1 Motivating scenarios

This section focuses on two scenarios that emphasize the need for self-configuration in a data storage system, as well as the specific adaptation requirements that cannot be addressed by generic scaling services, such as the Amazon *Auto Scaling*.

Elastic data-intensive applications. To run large scale applications in a IaaS Cloud, users typically need to deploy a distributed storage system on the provisioned virtual machines. The overall performance of such applications is directly affected by the configuration of the underlying storage system and by its fast data access capabilities. As it is essential to maintain the same application performance even when the size of the processed data sets increases, this scenario would benefit from a component able to reconfigure the backend system, according to some specific parameters.

Cost optimization. Although moving large-scale applications to the Cloud has become an undeniable trend, users of commercial Clouds have to take into account the new challenges introduced by these environments. Thus, optimizing the performance of applications is not enough, as an equally important concern is cost minimization. Cloud providers have imposed the pay-per-use model, where the users pay for the number of provisioned nodes and for the duration of their resource reservations. As a consequence, a means to reduce the costs of executing a specific system in a Cloud environment is to minimize the resource consumption. To reach this goal without experiencing any performance degradation, the system can be equipped with a component designed to inspect its behavior and scale in or out in correlation with internal events.

6.3.2 Global Architecture

The goal of the *Dynamic Configuration Framework* is to automatically optimize the utilization of resources in the system, while sustaining an optimal performance level.

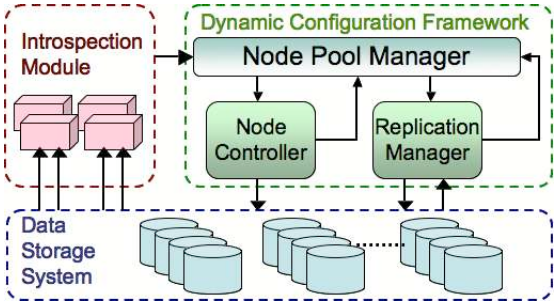


Figure 6.4: Architectural overview of the Dynamic Configuration Framework.

To this end, the system maintains two pools of storage nodes:

Active Nodes Pool (ANP): pool of currently active nodes that are used for data storage by the system.

Backup Nodes Pool (BNP): pool of nodes that are not employed by the storage system, but can be accessed and added to the system through specific operations.

The *Dynamic Configuration Framework* is designed to automatically switch storage nodes from one pool to another when certain conditions are met, in order to optimize resource usage. The decision to scale the node pool is based on retrieving the monitoring data and computing a heuristic score that evaluates the status of each storage node. The architecture of the *Dynamic Configuration Framework* is depicted in Figure 6.4. It consists of the following components:

The Node Pool Manager is the component in charge of making configuration decisions for the *Data Storage System*. It keeps track of the active storage nodes and automatically increases or decreases the size of the *Active Nodes Pool* according to specific parameters. It needs access to monitoring information describing the state of the system, and of the active data storage nodes. The implementation of *Node Pool Manager* is independent of the storage system it manages, as its input is only based on monitoring data retrieved from an *Introspection Module*. The *Dynamic Configuration Framework* can be configured through administrator-defined policies. Such policies define the parameters taken into account when enabling or disabling storage nodes and set thresholds for each of these parameters.

The Node Controller is responsible for enforcing the *Node Pool Manager*'s decisions by moving nodes from the *Active Nodes Pool* to the *Backup Nodes Pool*. This component represents the interface between the *Dynamic Configuration Framework* and the storage system. As a consequence, its implementation is dependent on the used system, as it has to enable/disable the data storage nodes and to let the system know of their state.

The Replication Manager has to fulfill two functions. First, it informs the *Node Pool Manager* about the replication requirements of the data stored on specific storage nodes. Each piece of data hosted by a *Distributed Storage Service* is replicated on a number of storage nodes, which is denoted as its *replication degree*. The second role of the *Replication Manager* is to maintain the replication degree of data residing on nodes that are about to be shut down by the *Node Controller*.

6.3.3 Dynamic scaling algorithms

The dynamic dimensioning mechanism implemented on the *Node Pool Manager* is presented in Algorithm 4. The *Node Pool Manager* periodically executes the DynamicScaling procedure. It receives a *scaling policy* defined by the administrator as a parameter to perform a customized tuning of the system configuration. The policy defines a set of parameters to be taken into account when expanding or reducing the system's node pool. It is structured as a list of key-value properties, denoting the following configuration elements:

- the names of system-wide parameters or node properties, such as the total load of the system, the size of the stored data or the number of accesses per storage node, which can impact the selection of the nodes to be added or removed from the system.
- specific sets of thresholds for each parameter.
- threshold weights corresponding to each threshold, to introduce a finer control over the significance of the parameter values.
- a weight value associated with each parameter to quantify the importance of the parameter in the decision-making process.

The decision making algorithm relies on the interaction with a set of primitives exposed by the other components in the *Dynamic Configuration Framework*. To access and process the monitoring data made available by the *Introspection Framework*, the *Node Pool Manager* employs two procedures:

EstimateNewANPSize. This procedure processes the scaling policy received as an input and inspects the current values of the parameters specified by the policy. It accesses the *Introspection Framework* to obtain up-to-date information and makes a decision aiming to optimize the number of nodes into the *Active Nodes Pool*. This decision is based on the set of system parameters specified by the policy and on their respective overall significance, modeled as weight values in the policy.

GetMonitoringData. This procedure contacts the *Introspection Framework* and returns the collected information corresponding to a set of nodes. This information is then processed according to the specifications of the scaling policy.

The *Replication Manager* has to expose a set of system-specific primitives that enable the *Node Pool Manager* to be aware of, and to control the replication degree of the stored data. The dynamic dimensioning algorithm makes use of the following procedures:

GetDataInfo. It returns the replication information corresponding to the data stored on a particular node, which is sent as a parameter.

RestoreReplDegree. Once a node is removed from the active storage servers, the *Replication Manager* is notified, as it is responsible for maintaining the replication degree of the data previously stored on the disabled node.

Since the *Node Controller* is in charge of the deployment of new nodes and the removal of unused nodes from the active servers pool, it interacts with the storage system by means of two primitives:

Algorithm 4 Dynamic Configuration.

```
1:  procedure DynamicScaling(ANP, policy)
2:      newSize ← EstimateNewANPSize(policy)
3:      if newSize < ANP.getCurrentSize() then
4:          for all nodes in ANP do
5:              monInfo ←invoke GetMonitoringData(nodes) on IntrospectionFramework
6:          end for
7:          sortedNodes ← monInfo.sort(nodes)
8:          toBeShutDown ← ANP.getCurrentSize() − newSize
9:          for all node in nodes andtoBeShutDown > 0 do
10:             score ← policy.computeScore(monInfo.getInfo(node))
11:             if score < scoreThreshold then
12:                 keep node in ANP
13:             else
14:                 toBeShutDown ← toBeShutDown − 1
15:                 nodeData ← invoke GetDataInfo(node) on ReplicationManager
16:                 disableNode ← True
17:                 for all dataID in nodeData do
18:                     if dataID.getReplDegree() < policy.getReplThreshold(dataID) then
19:                         disableNode ← False
20:                     end if
21:                 end for
22:                 if disableNode = True then
23:                     move node to BNP
24:                     invoke ShutdownNode(node) on NodeController
25:                     invoke RestoreReplDegree(nodeData) on ReplicationManager
26:                 else
27:                     keep node in ANP
28:                 end if
29:             end if
30:         end for
31:         return ANP.getCurrentSize()
32:     end if
33:     if newSize > ANP.getCurrentSize() then
34:         toBeStarted ← newSize − ANP.getCurrentSize()
35:         nodes ← invoke GetNewResources(node) on NodeController
36:         for all node in nodes do
37:             move node to ANP
38:             invoke DeployNode(node) on NodeController
39:         end for
40:         return ANP.getCurrentSize()
41:     end if
42: end procedure
```

SHUTDOWNNODE. This procedure removes a node (sent as an input parameter) from the *Active Nodes Pool*, that is it shuts down the running storage server and cleans up the stored data.

GETNEWRESOURCES. This procedure is used to obtain a set of new nodes by environment-dependent means. As an example, in the case of a system running on top of a Cloud platform, this procedure involves provisioning new virtual machines from the Cloud provider, deploying the required processes and adding the new instances to the system.

DEPLOYNODE. It is employed to deploy a storage server on new resources and to add it to the *Active Nodes Pool* of the system.

The dynamic dimensioning process relies on the results yielded by the EstimateNewANPSize procedure. If the estimated optimum size is less than the current size, then the system tries to disable the unnecessary nodes. To this end, it first has to determine which nodes are underutilized, i.e. for which of the active nodes the parameters defined by the scaling policy exceed the specified thresholds. This is achieved in two steps. First, monitoring data for each active node is collected from the *IntrospectionFramework*. Second, a score that reflects the value and the significance of each parameter is computed for each node. Next, the active nodes are sorted in the increasing order of their scores, where the highest scores designate the key nodes in the system that should not be removed (e.g. the nodes that store essential data or those that can sustain a large number of data access operations). For each low-score node, the system retrieves replication information about the data stored on the node from the *Replication Manager*. If the current replication degree of any of the stored data chunks is lower than a predetermined threshold, the node cannot be disabled without breaking the replication requirements for that specific piece of data. In the opposite case, the node can be moved to the *Backup nodes pool*. Moreover, the Shutdown primitive is invoked on the *Node Controller* and the *Replication Manager* is notified to restore the replication degree for all data chunks hosted by the nodes.

If the result returned by the EstimateNewANPSize procedure indicates the number of active nodes is too low to sustain an adequate performance level, then the pool of active nodes has to be extended with new resources. A request for new nodes is generated for the *Node Controller*, which is in charge of obtaining them by interacting with the environment. After the new resources are activated, the *Node Controller* is invoked to deploy and make them available to the system.

6.4 Summary

In this chapter, we proposed a set of self-management mechanisms targeted towards data-management systems. We first introduced a generic architecture for enhancing a system with introspection capabilities, relying on monitoring and processing system-specific parameters. Thus, a system can be considered self-aware, an essential step that opens the path to other self-management properties.

A second contribution consists in a generic security framework for defining and enforcing security policies. This framework can play the role of a self-protection component in a

data-management systems, being able to automatically react and provide customized feedback upon detection of security breaches. It allows administrators to define high-level policies for complex attacks and to detect and take appropriate measures to prevent harmful users from accessing the system.

Self-configuration is an essential property in the context of Cloud storage, enabling distributed systems to scale in or out whenever their real-time workload requires it. We proposed a self-configuration architecture aiming at optimizing the deployment scheme of a data-management system by transparently expanding or contracting the pool of running storage servers, according to various factors defined by the system administrator.

Chapter 7

Validation: Introducing self-management in BlobSeer

Contents

In the previous chapter we presented a set of self-management directions for data storage systems and we introduced generic frameworks to enable the self-* properties for such systems. This chapter focuses on enhancing BlobSeer, a distributed data-sharing system designed for highly-efficient concurrent data accesses, with self-management capabilities. Our goal is to build an autonomic, efficient and secure Cloud storage service by leveraging the generic self-adaptation frameworks and tuning them to the specific data-management protocols of BlobSeer.

7.1 Introspection mechanisms in BlobSeer

Adapting the generic introspection architecture to fit the design and implementation of BlobSeer requires careful consideration at several levels. First, it implies the identification of the relevant events that need to be monitored and of the aggregated data that can provide more compact information about the system entities. Second, the design of each layer of the introspection architecture plays a role in its overall performance and impact on the BlobSeer data-access operations. In this section we address these issues, by introducing the specific monitoring events that can be gathered from BlobSeer, and detailing the implementation of the introspection framework.

7.1.1 Collecting BlobSeer-specific data

This section investigates the relevant BlobSeer events that can effectively serve as input for various self-adaptation engines. We first detail the simple parameters that can be collected for each BlobSeer entity when a client requires access to data. Furthermore, we give an insight on the aggregated data that can be extracted from the basic parameters according to various goals.

The main monitoring data that can be gathered from the Blobseer entities can be summarized as follows:

Version manager. It handles client requests that focus on BLOB creation and publication of new versions when a WRITE operation is issued. Therefore it can generate two types of monitoring parameters:

- **BLOB creation.** A tuple containing the identifier of the client who initiated the operation (e.g. the IP address), the BLOB identifier, the number of needed BLOB replicas and the chunk size.
- **WRITE publication.** A tuple composed of the client identifier, the BLOB identifier, WRITE operation identifier, the assigned version, the offset and size of the published range of chunks.

Data providers. Each *data provider* is in charge of storing data chunks and transferring them to and from the client. Thus, two types of events have an impact on the performance of the *data providers*:

- **Chunk WRITE.** This operation entails the creation of a monitoring event consisting in a client identifier, BLOB id, chunk id, provider id and WRITE operation id. The latter value is crucial for tracking down all write requests for different chunks belonging to the same client WRITE operation, as well as for detecting the publication step of the same chunk range.
- **Chunk READ.** The monitoring data generated for this type of operation comprise a client identifier, BLOB id, chunk id and provider id.

With these basic parameters as a starting point, we define higher-level information that can be supplied to various self-adaptation components:

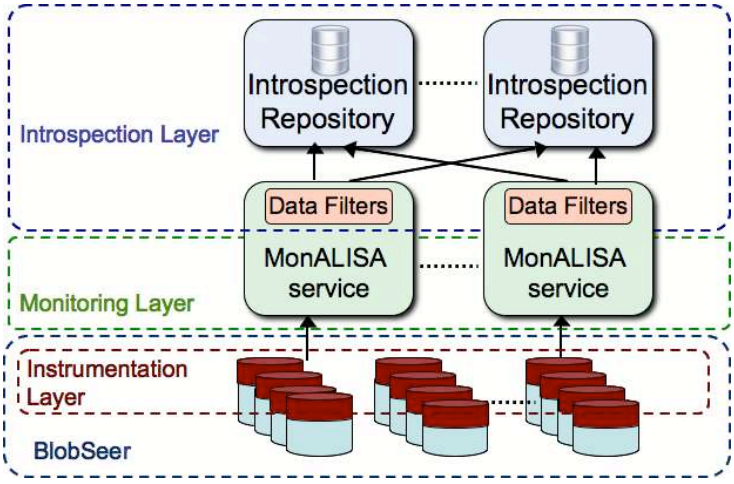


Figure 7.1: Architecture of the Introspection Framework.

Access patterns. This type of data includes the number of accesses per time unit for each BLOB, each BLOB version and each chunk. Such information can be generated by counting the number of write or read events collected for the data chunks and aggregating them according to the selected variable (BLOB, version, etc.).

User access history. The data yielded by the introspection layer can focus on the history of the user accesses to data. To this end, basic monitoring data can be aggregated into high-level illustration of each user's actions. For instance, such data comprise the number of written chunks for each WRITE operation or the number of read chunks in the last hour, for each user.

Provider load. The number of accesses at the level of each *data provider* represents a type of information needed by any self-adaptation component to assess the system load. The load of each *provider* per time unit can be obtained by computing the number of accesses for all data chunks stored on the *provider*. Furthermore, the *provider* load can be analyzed for each client, as a measure of the fairness of the data-sharing among clients.

Global state. This type of data can help self-adaptation components to make fast decisions when the system is heading toward a dangerous state. As an example the overall load or the total amount of free storage space play an important role in the BlobSeer system's capacity of providing an efficient data-access rate.

7.1.2 Implementation details

To enable the BlobSeer system to evolve towards an autonomic behavior, we adapted the 3-layered *Introspection Framework* proposed in Section 6.1, by designing specific interfacing blocks and plugging them on top of the generic components. Figure 7.1 displays the architecture of the *Introspection Framework* adapted for BlobSeer.

The lowest layer of the framework is the *Instrumentation Layer*, which is responsible for collecting data from the monitored system. Therefore, its implementation is closely linked

with the instrumented system, as well as dependent of the type of the gathered data. The *Monitoring Layer*, however, should not interfere with the targeted system and thus it can be built on top of any generic monitoring system able to address the following requirements:

Flexibility. As our goal is to optimize the performance of a distributed system through self-adaptation, it is essential for the monitoring system to accommodate both predefined and customized parameters. This feature allows introspection mechanisms to be aware of system-specific events and thus, to interpret the system state.

Scalability. BlobSeer is a storage system that deals with massive data, which is striped into a huge number of chunks scattered across the *data providers*. Moreover, typical access patterns include multiple clients simultaneously accessing various parts of the stored BLOBS. Each such data access generates a monitoring event; as a result, the monitoring system has to cope with a large number of storage resources and to efficiently manage high rates of monitoring events. Scalability also implies performance with respect to data collection, that is the response time of the monitoring layer has to be maintained regardless of the system load.

Reliability. System-specific monitoring parameters such as data accesses for each client are crucial events that need to be recorded by the monitoring system. Therefore, a desirable feature of the monitoring system is the reliability regarding monitoring data collection and storage.

Non-intrusiveness. A suitable monitoring layer should not degrade the performance of the data-access operations, that is the clients should not observe significant overheads when running data-intensive tasks on top of the monitored system.

The Monitoring Layer

To meet the aforementioned requirements, we built the monitoring layer on top of the MonALISA grid monitoring system, as it is a system designed to run in large-scale environments and to handle large amounts of monitoring events collected from distributed entities.

MonALISA (*Monitoring Agents in a Large Integrated ServicesArchitecture*) [61] is an event-based, scalable framework of distributed services, which provides the necessary tools for collecting and processing monitoring information. The essential components of the MonALISA framework are the *MonALISA services*, which are responsible for performing the data collection tasks. The typical data flow for the monitoring information within the MonALISA system is the following:

- The monitored entities generate monitoring events and send them to the MonALISA services.
- Each MonALISA service is equipped with a set of filters that select relevant events among the received data. Each filter processes the collected data and then relays them towards a local database or an external data repository.
- Monitoring data are persistently stored in repository databases and can be visualized by means of graphical clients provided by the MonALISA system.

The Instrumentation Layer

Apart from the monitoring services, MonALISA also provides an instrumentation library designed to monitor both a set of predefined parameters and various user-defined parameters for a given application. We used this library, called ApMon, to instrument the BlobSeer code and to send specific parameters to the MonALISA services in a distributed fashion.

The *Instrumentation Layer* comprises three modules that accomplish different tasks:

Generic parameters monitoring. This module uses ApMon to monitor the physical resources and to collect data such as the CPU load, available memory or network traffic. The MonALISA framework provides specific tools to monitor such parameters: ApMon can be enabled to automatically collect and forward them to the monitoring system through a background thread. The relevant event types can be defined in a configuration file, which also specifies the target MonALISA service.

Log parsing tool. In BlobSeer, the *version manager* is the entity that keeps track of all the BLOBS in the system and deals with the serialization of the concurrent requests. As it is essential to prevent any overhead when performing these tasks, the instrumentation module that monitors the *version manager* is implemented as a log parser able to send data to MonALISA. The instrumentation module scans the log file each time it is updated and reports the created BLOB or the written chunk ranges and their associated versions to a specific MonALISA service designated by the ApMon configuration file.

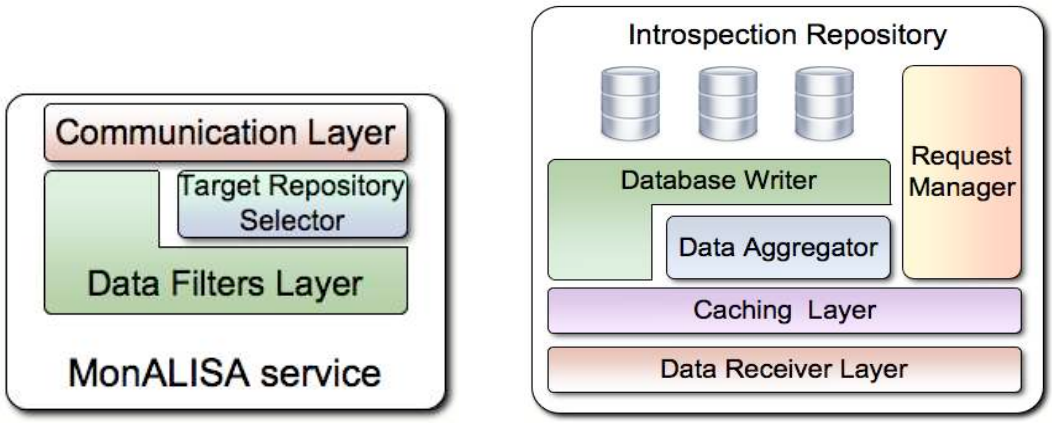
Instrumentation code. We equipped the BlobSeer entities with ApMon-based instrumentation code, so as to enable them to send monitoring parameters to the MonALISA services whenever an event occurs. Typically the instrumentation code has to specify the selected monitoring service (listed in a configuration file), the event type and the values of each field of the event.

The Introspection Layer

The MonALISA framework provides a *storage repository* for persistently collecting information from several monitored entities. The repository is backed by a relational database where it can automatically save the received data. However, the repository is optimized for time-series monitoring data and thus, for collecting generic parameters related to the physical resources. Our goal is to design an *Introspection Layer* that efficiently manages specific events, such as the ones corresponding to data accesses in a data management system, in order to use it as a building block for various self-adaptive components.

To this end, we developed the *Introspection Layer* as a set of modular services that focus on the following properties:

Efficient storage. The *Introspection Layer* is based on a set of *Introspection Repositories*, which are designed to store monitoring data in a distributed fashion. To enhance the performance of collecting monitoring data and load balancing within the *Introspection Layer*, each MonALISA service is equipped with data filters that partition the monitoring events and forward each subset to a particular *Introspection Repository*.



(a) The enhanced MonALISA service. (b) The Introspection Repository.

Figure 7.2: Implementation of the *Introspection Layer*.

Data aggregation. Each BlobSeer operation may generate a substantial number of monitoring events. Reading and processing all the collected data may slow down higher-level self-adaptation engines. To allow such self-adaptation components to efficiently analyze data and make decisions, we developed a module in charge of aggregating the gathered data. This approach generates compact data that represent specific properties, such as the access rate on data providers, the number of read operations on a BLOB version or the amount of data written by a certain user. Such data can then be easily fetched and interpreted by components that do not require all the details regarding a BlobSeer operation, but rather a concise summary.

Data caching. The rate of incoming monitoring data can considerably increase when a large number of users are concurrently accessing the system. Since each *Introspection Repository* is backed by a relational database, flushing the data to the persistent storage is typically a slow operation. To cope with this issue, each *Introspection Repository* implements a caching layer to handle the received monitoring events, while a dedicated process dumps them into the database.

The *Introspection Layer*'s implementation follows the diagram on Figure 7.2. It relies on a set of modules designed on two levels, which enable a customizable behavior of the *Introspection Framework*, according to the requirements of the upper layers. At the level of the MonALISA monitoring services, depicted on Figure 7.2(a), we placed the modules responsible for filtering the collected data and for sending them to the appropriate monitoring repository for persistent storage.

Data Filters Layer. The MonALISA system allows users to develop dynamically loadable modules to fulfill particular monitoring tasks. Such modules can be used to filter the incoming monitoring data, to process it and periodically forward it to other components. We designed a set of filters for each BlobSeer-specific event. Their goal is to parse the events, to identify their type and to accordingly select an appropriate *Introspection Repository* where the data can be stored.

Target Repository Selector. To optimize load balancing between the distributed *Introspection Repositories*, we designed a *Selector Layer* that picks out the repository by means of configurable rules. The default selection strategy we used is represented by a hashing function that partitions the data according to the user who generated it.

Communication Layer. To provide efficient communication between the MonALISA services and the *Introspection Repositories*, we implemented an asynchronous *Communication Layer*, which transparently handles the communication details, such as socket management and data transfers. It is implemented on top of the NIO Java library, a collection of Java APIs for intensive I/O operations.

The architecture of the second level of the *Introspection Layer*, namely the *Introspection Repository*, consists of several modules backed by a relational database, as shown in Figure 7.2(b).

Data Receiver Layer. This layer can handle multiple concurrent requests to store monitoring data in an asynchronous fashion. It collects the received data and forwards them to the appropriate cache.

Caching Layer. It is in charge of temporary hosting the received data, so as to enable the *Introspection repository* to deal with high rates of received monitoring events.

Data Aggregator. To meet the needs of higher-level components, this module can implement various aggregation strategies. Basically, it scans the monitoring tables in the repository database and creates new tables that store coarser-grained data that enable self-* mechanisms to observe global patterns and trends.

Database Writer. The implementation of the *Introspection Repository* includes a dedicated process for flushing the cached monitoring events into the backend database. It is also in charge of marking the events that have already been persistently stored, so that they are safely removed from the cache when new data arrives. The module can be interfaced with various database systems, as it relies on an extendable storage interface.

Request Manager. This component is designed to increase the performance of remote requests for monitored data, by caching the most recently collected events and aggregated values. It serves requests by first looking into the cached events and retrieving the results from the backend database only if they cannot be found into the cache. The module can be customized with respect to the size of the cache and the type of cached data.

7.2 The security framework

We validated the *Security Management Framework* we proposed in Section 6.2, by interfacing it with the BlobSeer system. It serves as a self-protection component able to detect malicious attacks in Cloud environments. This section describes the types of attacks that can impact on data-access performance in BlobSeer. Next, we give an insight on the our framework's policy definition language by focusing on a critical case study scenario: Denial of Service attacks. Finally, we present some implementation details for the main components of the *Security Framework* and their interactions with BlobSeer.

7.2.1 Security attacks in BlobSeer

To evaluate the *Security Management Framework*, we identified a set of security vulnerabilities in BlobSeer and we modeled security policies to target the associated attacks. The key malicious behavior scenarios that have to be prevented in order to maintain the high-throughput client operations in BlobSeer are summarized as follows:

Protocol breaches

A malicious user can try to compromise the system by deliberately breaking the data-access protocols. In BlobSeer, the most vulnerable operation is data writing, as it exposes several targets for malicious users: first, it consists in inserting new data into the system, which can be a means to overload *data providers*; second, writing data to BlobSeer implies generating new BLOB versions, operation that can lead to data inconsistencies.

The WRITE operation imposes a strict protocol to the user that wants to correctly insert data into the system, as illustrated in Chapter 5. It consists of two independent phases that have to be executed consecutively:

The data-writing step comprises the user's request for a list of *data providers* and the transfer of the chunk range to be written to those *providers*.

The data-publication step comprises the creation of the metadata associated with the written data, and the publication of the written chunk range as a new version.

A correct WRITE operation is defined as the successful completion of the aforementioned steps, with the constraint that the published information concerning the written chunk range is consistent with the actual data sent to the *data providers*. The WRITE primitive implemented by the BlobSeer client library is designed to carry out a complete and correct WRITE operation. However, as the code is open source, a malicious user may modify the library so as to break the WRITE protocol. As a consequence, there are three types of protocol breaches that can be detected for the WRITE operation:

Data written and not published. In this case, a malicious user obtains a list of *providers* from the *provider manager* and then starts writing data to the *providers*. The second step is never issued and thus the *version manager*, which keeps track of all the BLOBS and their versions, will never be aware of the data inserted into the system.

Data published without being written. This scenario is representative for a user who attempts to compromise the system by making available data that does not actually exist. Thus, other users might try to read the published data without being aware that the metadata contain fake references.

Inconsistencies between the written and the published data. The attack that corresponds to this situation aims to disrupt the computations that use data stored into the BLOBS. As an example, the user might only write the data corresponding to the beginning of the published range. Therefore, an application can start reading and processing the data and discover only later that the current BLOB version is incomplete. Hence the computation would be compromised and the application forced to restart the processing.

```
<?xml version="1.0" encoding="UTF-8"?>
<securityPolicy id="1_47">
    <clientID rvalue="c" value="c" />
    <active value="true" />
    <priority value="1" />
    <start value="w1" />
    <end value="c1" />
    <preconditions>
        <event id="w1" type="prov_write_summary">
            ...
        </event>
        <event id="p1" type="vman_write">
            ...
        </event>
        <event id="c1" type="check">
            <content value="wsc > twsc" />
        </event>
    </preconditions>
    <enforcement>
        ...
    </enforcement>
</securityPolicy>
```

Figure 7.3: Security policy for Denial of Service on data providers.

Denial of Service attacks

In order to preserve the transfer throughput level and a fair bandwidth sharing among concurrent users, the BlobSeer system has to prevent Denial of Service attacks at various levels. Since the BlobSeer entities fulfill complementary functions, a user may contact each of them during a legitimate action. Attempts to flood the *version manager* or the *provider manager* have to be detected and identified as Denial of Service attacks, as they may prevent other users from accessing the system.

Multiple concurrent requests targeted at *data or metadata providers* are, however, more difficult to handle, as such actions represent the first stage of any valid READ or WRITE request. In this case, the administrators need to design more complex security policies able to define the limits between what is considered normal behavior and malicious operations aiming at disrupting proper system functioning.

Abnormal client activity

Whereas the first two scenarios emphasize misbehavior detection, another approach is to analyze the user's access patterns and to identify anomalies in its activity, even though it is correct with respect to the data access protocols. In this case, a deviation from the previously observed behavior can be a symptom of an unauthorized access to data or an attempt to affect the system. As an example, an authorized user may try to read all the BLOBS it has access to. While such an action is legitimate, it can be identified as a *crawling* attack, especially if the history of the user actions does not contain similar requests. This type of attacks have to be detected and recorded, as they may influence the *trust level* corresponding to the user who initiated them. In Section 6.2, we introduced the *trust level* of a user as a score computed by the *Trust Management* module of the security framework, which assesses the behavior of the user based on its past actions.

7.2.2 Case study: DoS attacks in BlobSeer

This section details the structure of Denial of Service attacks specific to BlobSeer. Furthermore, we focus on attacks targeted at *data providers*, providing an in-depth description of a security policy used to identify such attacks.

Denial of Service attacks are characterized by a large number of request aiming at disrupting the normal execution of a system and preventing other clients from performing legitimate operations. In the case of BlobSeer, each valid data-access operation implies a series of parallel requests and data transfers. For instance, a single WRITE operation may comprise hundreds of chunk transfers, especially if the chunk size is small. Consequently, a Denial of Service attack attempted by a user who simulates a regular operation can be more difficult to detect than a simple flooding attack that generates random requests to overload a service.

We consider the former approach, as it deals with vulnerabilities inherent to BlobSeer and it represents a typical illustration of a Denial of Service attempt. Defining a security policy for such an attack allows us to easily derive policies for the other simpler forms of Denial of Service targeting BlobSeer entities.

As detailed in the previous section, a typical WRITE operation in BlobSeer consists in: (1) writing a set of data chunks to the *data providers*; and then (2) publishing the write as a new version of the BLOB on the *version manager*. We define a Denial of Service attack based on the WRITE primitive in BlobSeer as a valid WRITE operation where the number of written chunks within the same operation is larger than a predefined threshold. In other words, the amount of data written by a user before issuing a new version has to be below a specific limit that ensures that no user is able to overload specific data providers or the whole system.

Figure 7.3 illustrates a representative security policy corresponding to a Denial of Service attack on *data providers*. The policy is represented through an XML language, using tags that follow the structure introduced on Figure 6.3. The goal of this policy is to capture all writes on *data providers* that comply with the following rules: they have been performed in a specific time interval, they belong to the same WRITE operation and they have not been published by the end of the time interval. If such a behavior is detected in the collected activity history for a specific user, then the user is identified as being malicious and a set of feedback actions are injected into the BlobSeer system.

The top-level XML element of the policy lists the *General Parameters*. In this case study, it states that this policy has a high priority and will be applied to a specific client, identified at runtime. The *Preconditions* tag encloses a list of three event types that play a role in a DoS attack. We identify start event *w1* that models a WRITE operation and event *p1* that denotes a publication operation. The final event *c1* verifies if the thresholds associated with the matched events have been exceeded. When the template events in the policy are filled in with real monitoring values and thus an attack is detected, the *Security Framework* scans the *Enforcement* tag and identifies the feedback actions to be forwarded to the system.

Figure 7.4 shows the contents of start event *w1*, which identifies the WRITE operations on the *data providers*. To select relevant data from the *User Activity History*, the *Properties* tag defines a set of specific elements associated with a WRITE operation: they include the BLOB id and the operation watermark (which identifies the WRITE operation in BlobSeer), but also the number of written data chunks (i.e. *NoWritesCount*). The latter is the key element that helps defining the attack. It has a corresponding threshold, denoted *thresholdNoWrites*.

```
<event id="w1" type="prov_write_summary">
    <clientID rvalue="" value="c" />
    <properties>
        <blobId id="bId" rvalue="" value="b" />
        <watermark id="wa" rvalue="" value="w" />
        <NoWritesCount id="wsc" rvalue="" />
        <thresholdNoWrites id="twsc" value="100" />
    </ properties>
    <timeFrame>
        <firstTimestamp id="fts" rvalue="" />
        <distance id="dist" value="7000" />
    </ timeFrame>
    <continuous>
        <refProperties value="wsc" />
    </continuous>
    <neg>
        <followedBy>
             <refEvent value="p1"/>
             <distance value="<= fts + dist"/>
        </followedBy>
     </neg>
</event>
```

Figure 7.4: The structure of an event for a DoS security policy.

```
<enforcement>
    <rule>
        <constraints>
            <and>
                <content value="TL < 70" />
                <content value="TL > 30" />
            </and>
        </constraints>
        <actions>
                <enablePolicy value="20_9" />
        </actions>
    </rule>
    <rule>
        <constraints>
            <content value="TL < 30" />
        </constraints>
        <actions>
            <blacklist rvalue="c" value="c" />
        </actions>
    </rule>
</enforcement>
```

Figure 7.5: Enforcement rules for a DoS security policy.

The event further contains a *TimeFrame* tag used to position the WRITE operation in time. This is done by recording the timestamp of the first detected chunk write into the *firstTimestamp* element and by specifying a *duration* for which the policy is supposed to search for malicious user actions. As the policy has to capture all write events, the *continuous* tag is employed in the event's structure to specify which parameters may vary among the matched writes: the number of written chunks, which increases each time a new write is found.

The event listing defines as well the correlations with other events needed to instantiate the policy. In this scenario, the write event must not be followed by a publication operation (modeled by event *p2*) by the end of the time interval delimited by the timestamp of the first write event recorded, denoted *firstTimestamp*, and the *duration* tags.

The final part of the security policy comprises a set of enforcement rules presented in Figure 7.5. When the policy is matched for a specific user, the rules are sent to *Policy Enforcement* module. Each rule comprises a set of *Constraints* and *Actions*. The *Policy Enforcement* module verifies which constraints are satisfied and then it executes the corresponding actions. The constraints are typically employed to customize the feedback suggested by the policy, according to the *trust level* of the user. The actions range from logging the attack in the history database (in the case of minor attacks) to activating new security policies and blacklisting the user (when the attack has an important impact on the system and the user has a low *trust level*).

7.2.3 Implementation details

The implementation of the *Security Management Framework* relies on a set of modules following the structure presented in Section 6.2.

Policy definition

A security policy can be submitted to the *Security Violation Detection* module as an XML file structured in the format presented in the previous section. Each security policy comprises a set of mandatory elements, corresponding to the structure defined in Section 6.2.3: general parameters, preconditions, enforcement rules. The pattern of the considered attack is modeled by the *Preconditions* element, which contains a list of template events. Each event has in turn an extensible format that complies with the generic structure of the event.

Complex attack scenarios may require security policies that include many events interconnected through logical operations. To simplify the policy writing task for administrators, the *Policy definition* module exposes an API that can be used to define security policies. This module provides a set of interfaces modeling the main elements of a security policy, designed to facilitate the generation of the policy XML files.

Malicious users detection

The goal of the *Security Violation Detection* component is to apply the policy breach detection algorithm for all policies registered in the system and for all the users that access it. To achieve this goal, the implementation relies on the following building blocks:

Multithreaded execution. To verify multiple security policies in parallel, the detection component is designed as a thread pool that assigns a thread to each new policy that has to be processed. When the policy matching can no longer proceed (e.g. when the detection process has to wait for monitoring data), the changes made to the initial policy are stored as a *partially matched policy* and the thread is returned to the pool. Each type of policy is periodically checked, according to a *priority mechanism*. Its implementation allows the policy breach detection algorithm to verify critical policies with a higher rate than the policies that do not have a significant impact on the system's response to the user.

Dynamic policy loader. The system has to be aware of both the template policies and the *partially matched* ones. To this end, the *Security Violation Detection* component includes

a module that scans particular locations on the file system to discover existing policies. It is able to load any policy described as an XML file and forward it to a worker thread. Furthermore, this module provides fault tolerance by saving the *partially matched policies* as XML files and reloading them in case of failures, along with the template policies. This mechanism is also important for policies describing complex attack patterns, which take a long time to manifest themselves. In such a case, the execution thread saves the *partially matched policy* as a file that will be reloaded at a later point in time, according to its priority, and the algorithm resumes its execution. The tradeoff for storing all *partially matched* policies is that some of them will never be matched (i.e. no attack is detected for that specific client) and they have to be garbage collected to prevent the framework to load them again.

User History Management. The *Security Management Framework* can be used for various systems that comply with a unique requirement: the targeted system has to collect relevant user-generated events into a *User Activity History*. The *User Activity History* is typically implemented as a database recording user identification information, generated event types and specific event properties. In order to provide security mechanisms specifically tuned for BlobSeer, we used the introspection component to generate the *User Activity History*, as it is able to generate all the relevant data needed by the security framework, while remaining non-intrusive. The user actions are recorded into a database that includes both the users past activity and the information monitored from their current operations. As an example, for publishing a WRITE operation in BlobSeer, the database may store the user IP address, and the BLOB id, version, data size and offset.

Dependency-checking engine. Each policy is defined as a set of interconnected template events. To detect the attack targeted by the policy, the security framework has to learn the event dependency graph, to fill in each event with real user events and to parse the subsequent events. We designed a *Dependency-checking engine* to accomplish the two following tasks: first, to parse the policy and create the event dependency graph; second, to evaluate the logical constructions that link two adjacent events to determine whether the detected flow of events is a match for the malicious behavior defined by the policy.

For instance, a simple protocol breach attack in BlobSeer can be modeled as an undefined number of chunk write operations, which are *not followed* by a *publish* event for the same range of data within a specific *Timeframe* (i.e., an attack for which data is written to the data providers, but it is not published). The goal of the *Dependency-checking engine* in this case is to validate the attack if the *Security Violation Detection* component discovers a user that performed a series of chunk write operations, but it cannot fill in the *publish* event with real data in the given *Timeframe*.

Query creation. The policy breach detection algorithm has to submit a query to the *User History Management* module each time it tries to detect monitored events that match a template event defined by the policy. The *Query creation* module handles the creation of the database queries complying with the specific requirements of the needed events. These requirements include: building the database table names specific for the queried event type, selecting the required fields from the database according to the *Properties*

defined within the event and adding query constraints related to the event's *Timeframe* and *Properties* thresholds.

Task partitioning. The security framework attempts to instantiate each template policy for every user in the system; furthermore, *partially matched policies* are associated with specific users. As the detection mechanisms for different policies do not interfere at any level, the *Security Management Framework* is designed to be replicated on several servers in charge of a fraction of the total number of users. This approach enables a faster attack detection in the case of a large number of clients concurrently accessing the system.

Policy enforcement

When all the template events of a security policy have been filled in with real values from the monitored user events, the set of enforcement *Actions* and *Constraints* specified by the policy are forwarded to the *Policy Enforcement* module. It verifies the *Constraints* and selects the *Actions* corresponding to the satisfied constraints. A key type of constraints are the ones related to the user's *trust level*. This mechanism allows the *Policy Enforcement* component to customize the feedback sent to the system. It imposes more severe measures for untrusted users and applying only mild penalties for users who do not have a history of detected malicious actions.

In the case of BlobSeer, we consider two types of feedback actions, which are predefined in the policy-definition API:

Record the attack in the history database. This is the default operation performed for each detected attack, as this information can provide a compact summary of a user's behavior. The history of the user's previous malicious actions is an essential input for the *Trust Management* module, as well as for complex security policies that need to correlate user events with past attacks.

Execute custom scripts. To provide a flexible feedback mechanism, the implementation of the *Policy Enforcement* component enables administrators to specify custom scripts as reactions to detected attacks. This approach allows a generic system implementation capable of providing a feedback response tailored to the system needs. Such scripts include sending alerts to administrators, setting thresholds on the maximum data size for each client or preventing the client from accessing the system.

To enable self-protection in BlobSeer, we implemented a mechanism to prevent malicious users from writing data into BlobSeer, which consists in a modified *provider manager*. This is the entity that enables a BlobSeer user to write new data into the system, by supplying it with a list of *data providers* that can store the new chunks. We extended the *provider manager* to store the set of users that have performed malicious attacks and have to be banned from accessing the system. When a blacklisted user intends to perform a WRITE operation and requires a list of *data providers*, it will receive an error response from the *provider manager*; this mechanism prevents users from further proceeding with their actions, thus denying them the WRITE access to data. This approach, however, cannot restrict users' read access to data, as this operation starts with a request targeted at the *version manager*. Moreover, if the

malicious users have already cached the metadata and the *data providers* storing data they are interested in, they can skip the first phase of the any data access operation and avoid the blacklisting mechanism. To fully isolate malicious clients, a similar blacklisting module can be implemented on top of each BlobSeer entity: the *provider manager* keeps track of existing providers by employing a heartbeat mechanism [75]; the same approach may be used to disseminate the blacklist updates to the *data providers*.

Furthermore, we enriched the API exposed by the *provider manager* with two primitives allowing administrators to blacklist malicious users, or to remove specific users from the blacklist. Thus, the BlobSeer administrator can configure the *Enforcement* element of a security policy to execute a script that contacts the *provider manager* and indicates a user to be blacklisted.

Trust Management

The *Trust Management* module represents the link between the dynamically changing state of the data management service and the *Policy Management* module. This module associates a *trust level* to each user, thus enabling the system to customize the feedback of any detected policy violation by taking into account the history of each user.

The *trust level* is computed as a score for each user, based on the following input: the history of malicious actions, the system state and the type of detected attacks. The *Trust Management* module tracks the system state by analyzing a set of collected monitoring data. The system state is influenced by the density of requests, but also by the load and available memory of the different entities in BlobSeer. The *User Activity History* is implemented as a database in BlobSeer, and it is accessed by the *Trust Management* module through the same *User History Management* module employed by the attack detection components. Besides user activity events, the database has to include aggregated data describing physical parameters collected from the BlobSeer nodes.

The strategy employed to compute the *trust level* is configurable, and the administrator can implement a wide range of *Trust Policies*, relying on the history of a specific user actions, the consequences of the detected attacks or the state of the system. To validate our *Trust Management* module, we implemented a simple calculation formula for the *trust level*, which takes into account the following three parameters:

$$trustLevel = \sum A_i \times Age(time_i) \times SystemState(time_i) \tag{7.1}$$

Parameter A_i is a numeric coefficient associated with each type of attack described by a security policy. Its goal is to quantify the significance of the attack and its potential impact on the overall performance. Its value is fixed for a given policy and it is stored in a table that lists all possible attacks. The second coefficient, $Age(time_i)$, is a time function, used to give a higher weight to more recent events.

$$Age(time_i) = \frac{1}{currentTime - time_i} \tag{7.2}$$

In this formula, the *currentTime* is the time when we compute the *trust level* and $time_i$ the time when the attack was detected. This approach guarantees two properties of the *trust level*: first, it enables the *trust level* to increase when the user performs only valid operations;

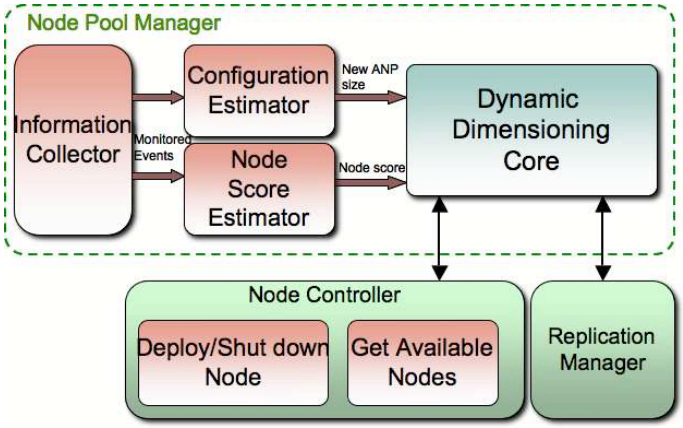


Figure 7.6: Architecture of the Dynamic Configuration Framework.

second, the system can quickly react to incoming attacks, as they are immediately reflected in the decreasing value of the *trust level*.

The *SystemState(i)* is a function that returns a value describing the system state at the moment when the user attempted the attack. We incorporated this value into the formula to adjust the *trust level* according to the impact that each action has on the system. For instance, a Denial of Service attack may be issued when the system is under a heavy load. Such an attack will be associated with a higher penalty with respect to the *trust level* than if it were attempted at a time when the system was under normal load, as it may have a major impact on the performance of the system.

7.3 Self-configuration in BlobSeer

To provide support for self-configuration in BlobSeer, we rely on the *Dynamic Configuration Framework* introduced in Section 6.3. This section shows how we integrated the auto-scaling mechanisms into BlobSeer and it gives an insight on the implementation of the BlobSeer-specific components.

7.3.1 Dynamic configuration in BlobSeer

In order to adapt the *Dynamic Configuration Framework* for the BlobSeer system, we designed a set of interface modules depicted in the red blocks on Figure 7.6:

Configuration Estimator. This is the core of the *Node Pool Manager*, which is responsible for estimating the size of the *Active Nodes Pool*. The decision process implements a configurable dimensioning strategy, based on the monitoring information received from BlobSeer and on the contents of the scaling policy (i.e., the policy that describes the behavior of the *Dynamic Configuration Framework*, as defined in Section 6.3).

Node Score Estimator. This module computes a score associated with an active *provider*, taking as an input the scaling policy, which details the parameters that make up the score. The implemented strategy can be configured to take into account various factors or past events monitored by the *Introspection Framework*.

Information Collector. This module requires that Blobseer is equipped with the *Introspection Framework*, which enables it to access real-time aggregated information concerning the providers load. The persistent storage of the introspective information is represented by the database that backs up the *Introspection Repository*. The *Information Collector* needs to be aware of the structure of the database and to implement specific queries that yield the parameters required by the *Configuration Estimator*.

Node Controller. The only BlobSeer entity that keeps track of the number of active *providers* is the *provider manager*, which employs a *hearbeat mechanism* to discover the failed *providers*; moreover, each newly deployed *provider* registers itself with the *provider manager*. As a consequence, the *Node Controller*'s only task is to deploy and to shut down BlobSeer *data providers*, as the system can take care by itself of integrating the new *providers* into the data flow or of removing the inactive ones. This process mainly depends on the environment where the BlobSeer system is running. As an example, on a local cluster that hosts BlobSeer on a subset of its nodes, the *Node Controller* can be implemented as a simple script that turns off processes on the unneeded nodes. On another hand, it has to keep track of the available nodes in the cluster, in the form of a *Backup Nodes Pool*, and to deploy the BlobSeer *providers* on some of these nodes, which then migrate to the *Active Nodes Pool*.

To validate the BlobSeer *Dynamic Configuration Framework*, we implemented a straightforward strategy to compute a score for the active *data providers*. The *data providers* that correspond to the highest scores are the ones proposed for removal from the system and the *Active Nodes Pool*. Indeed, a high score reflects the low significance of a node for the overall system performance with respect to the configured factors. We considered two types of factors that influence the importance of each active *provider* in the system:

Physical factors. The free disk space, the average bandwidth usage and the CPU load.

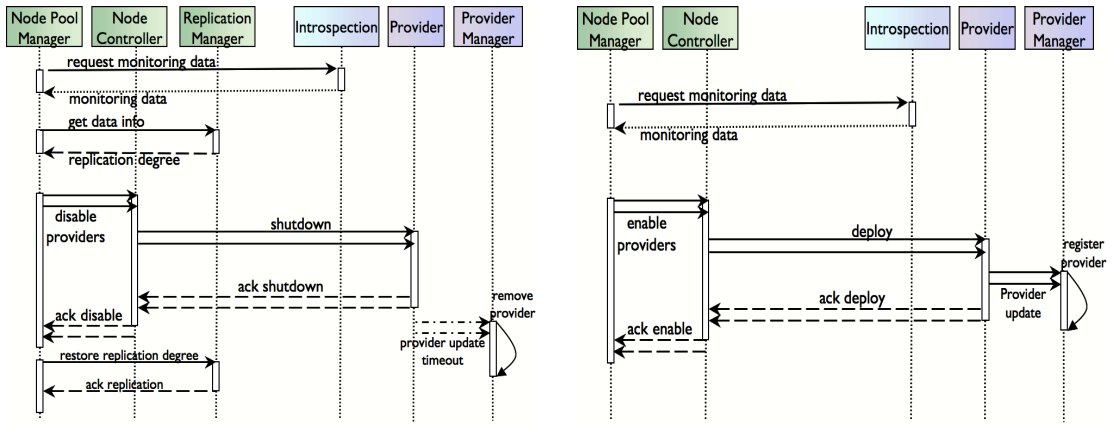
BlobSeer factors. The number of read/write accesses per time unit and the size of the stored chunks.

The *Configuration Estimator* relies on a simple score computation formula that takes into account the set of factors and corresponding weights and thresholds:

$$score = \sum_{i=1}^{n} wf_i \times thw_i \tag{7.3}$$

In this formula, wf_i represents the weight of the factor i, as defined in the scaling policy. For each factor, the policy also specifies a set of thresholds and associated *threshold weights*, modeled through the thw_i variable. We introduced the *threshold weights* to handle the influence of various factors in a more fine-grained manner.

As an example, the *threshold weight* of the free disk space (and therefore the score) is higher if the monitored value of the factor is more than 50%, so as to increase the chances that the node is selected to be shut down. However, if the free disk space is more than 90%, the associated *threshold weight* is greater: thus, the score can reflect the negligible amount of data stored on that particular node, promoting it as a more relevant candidate for the *Backup Nodes Pool*.



(a) Reducing the number of data providers. (b) Increasing the number of data providers.

Figure 7.7: Dynamic Configuration Framework: interaction with the BlobSeer entities.

With this formula as a starting point, our approach compares the scores with an overall threshold specified by the policy. This threshold should represent the value above which a provider is no longer useful to the system, for instance when it stores only a negligible amount or data or it represents a bottleneck for data-access operations. This value can be evaluated by taking into account the history of the scores computed for the running providers under various system loads and deployment configurations. The *Configuration Estimator* employs the overall threshold to estimate the new size of the *Active Nodes Pool*, as being the number of data providers that obtained a score lower than the threshold. The new size is reported to the *Node Pool Manager*, enabling it to proceed with the system reconfiguration.

The interaction between the *Dynamic Configuration Framework* and the BlobSeer entities is depicted on Figure 7.7. This sequence diagram illustrates the requests coordinated by the *Node Pool Manager* for decreasing and increasing the number of the active *providers* used by the BlobSeer system. The *Node Pool Manager* only interacts with the *Introspection Framework* and the *Node Controller* directly manages *provider* removal or deployment. The operation is transparent for the other BlobSeer entities, excepting the *provider manager*, which, however, is able to automatically detect the modifications and to update its internal list of *providers*.

7.3.2 Zoom on replica management

The *Replication Manager* module is in charge of maintaining the replication degree of the data chunks stored on the disabled *providers*. The reliability of the *Dynamic Configuration Framework* lies in its capacity to transparently reduce the number of data providers without affecting the stored data. We designed a *Replication Manager* module that takes advantage of the efficient data management techniques in BlobSeer. The key BlobSeer features that directly influenced the architecture of the *Replication Manager* are detailed as follows:

Immutable data and metadata. In BlobSeer, a data chunk is never overwritten; each overlapping WRITE operation generates new data chunks instead, creating a new BLOB version. Furthermore, BlobSeer creates a new metadata tree associated with each BLOB

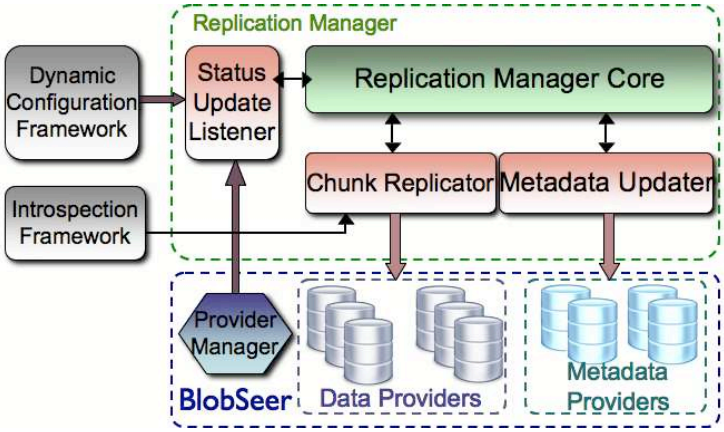


Figure 7.8: Architecture of the Replication Manager.

version, whose leaves contain the addresses of the *data providers* that host the chunks. As a consequence, there cannot be any concurrent write accesses on either data or metadata. This feature is crucial for the design of the *Replication Manager*, as there is no need to track or maintain replica updates, and, moreover, all replicas of a chunk of data are consistent at any time.

Generic data and metadata providers. The implementation of the *data and metadata providers* relies on generic key-value stores that can be accessed through RPC calls. This enables the *Replication Manager* to contact any *provider* and to create new replicas through its standard API, in a transparent manner.

The dynamic replication process relies on the following steps:

Replication request. The *Replication Manager* listens for replication requests from the *Dynamic Configuration Framework*, which typically include a set of chunks that need to be replicated.

Create chunk replicas A dedicated module interacts with the *Introspection Framework* to discover the *providers* that store each chunk and a set of available *providers*. The chunk is read and replicated to another *data provider*, through a standard write call on the *data provider*.

Update metadata. The *Replication Manager* reads the metadata tree associated with the BLOB version for the given chunks. It then modifies the contents of the tree leaves corresponding to the chunks: for each chunk, it updates the list of chunk replicas hosted on its corresponding leaf.

Implementation details

The architecture of the *Replication Manager* module presented in Figure 7.8 is based on the following components:

Status Update Listeners. These components are in charge of collecting the replication requests. Such requests can be generated by one of the following modules:

- The *Dynamic Configuration Framework*, when it needs to shut down some of the running *data providers*.
- The *Provider Manager*, when it detects that a provider has failed.
- A dynamic replication engine that can analyze monitoring events from BlobSeer and automatically tune the replication degree of the data chunks according to their access rate.

Chunk Replicator. The first step towards the replication of a data chunk is copying it to an available *provider*. The *Chunk Replicator* performs a set of steps similar to the ones of a typical WRITE operation. First, it contacts the *provider manager* to obtain a list of *data providers* that can store the new replicas. As the selected *providers* must not host other replicas of the same data, the *Chunk Replicator* needs access to the *Introspection Framework* to verify the contents of the *providers*. It can be configured to repeat this first step until it obtains suitable *providers* or until it reaches a specified number of retries. Second, the *Chunk Replicator* writes the chunks on the selected *data providers* using the same RPC mechanism employed by the BlobSeer entities [75].

Metadata Updater. Once the data chunks are written on the providers, the *Chunk Replicator* has finished its task. It forwards the chunk identifiers and the *data providers* where they are replicated to the *Metadata Updater* module. It is responsible for making the replicas available to the BlobSeer clients. To this end, it has to modify the information stored by the metadata trees: for a specific version of a BLOB, each metadata leaf corresponds to a data chunk and it stores a list of replicas for that chunk. The *Metadata Updater* requests the metadata tree root from the *version manager*, it scans the tree to find the leaf associated with a particular chunk, and finally it updates the list of existing replicas, by adding the new replicas and removing the unavailable ones. This mechanism is possible due to the design of the BlobSeer system, which avoids concurrent write attempts on the metadata; once the BLOB version is published, no BlobSeer entity will subsequently modify the metadata, allowing the *Replication Manager* to access it in an efficient and consistent manner.

7.4 Summary

In this chapter we provide a real-life use case for the self-management frameworks proposed in Chapter 6. We show how we enhanced BlobSeer, the versioning-based data-management system introduced in Chapter 5, with three self-* properties: self-awareness, self-protection and self-configuration. Additionally, we give an insight of the implementation of the self-* mechanisms and of the required interface modules that enabled the interconnection with BlobSeer.

Chapter 8

Evaluation and results

Contents

In the previous chapter, we validated the introspection framework and the generic self-adaptation architectures by integrating them with the BlobSeer data management system. The integration mainly focused on the implementation choices and on the design of the interface modules that enable the interaction with BlobSeer. This chapter first introduces Grid'5000, the experimental testbed we used. Then, it presents the deployment platform that allowed us to perform large-scale tests involving various self-adaptation frameworks. Furthermore, we provide a set of experimental results that assess the performance of the introspection framework and we evaluate the self-protection BlobSeer component through synthetic concurrent attack scenarios.

8.1 Experimental testbed: the Grid'5000 platform

The Grid'5000 [54, 3] project is a research effort aiming at developing a large-scale testbed for parallel and distributed computing research. The Grid'5000 infrastructure provides researchers with a highly-configurable environment, which allows users to perform reproducible experiments under real-life conditions for all software layers ranging between network protocols up to applications.

The platform is geographically distributed over 10 sites in France: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Reims, Rennes, Sophia-Antipolis and Toulouse. Two foreign sites (Luxembourg and Porto Alegre in Brazil) have recently joined the Grid'5000 project, which now features more than 7000 CPU cores.

8.1.1 Infrastructure details

Grid'5000 was developed as a hierarchical infrastructure, where the computing nodes are grouped into clusters and several clusters form a site.

Resources

Grid'5000 is designed as a federation of independent clusters and therefore it consists of a complex hierarchy of heterogeneous physical resources. The resource heterogeneity concerns various levels of the architecture, as detailed below [3] :

Processor. The processor families include AMD Opteron (62%) and Intel Xeon EMT64 (32%), featuring mono-core (38.5%), DualCore (32%), QuadCore (27%), 12-core (2.7%) processors.

Memory. The nodes are equipped with at least 2 GB of physical memory, which can increase up to 48 GB for some clusters, such as *parapluie* on the Rennes site.

Network. The network interconnects range from Ethernet cards to high-speed Infiniband or Myrinet links for intra-cluster communication.

Although all nodes in Grid'5000 are equipped with Unix-based operating systems, the distributions vary across sites, as well as the set of pre-installed libraries. This issue has to be taken into account for multi-site deployments, as user applications may require specific libraries and tools.

Network

The various sites are interconnected through a high-performance backbone network infrastructure provided by RENATER, the French National Telecommunication Network for Technology, Education and Research. The architecture is based on 10 Gbit/s dark fibers and provides IP transit connectivity, enabling inter-site latencies in the order of 10 milliseconds. Within each site, the resources are typically interconnected through Gigabit Ethernet switches. Some sites also provide high speed and low latency interconnects, such as Myrinet or Infiniband, which feature 10 Gb/s and 20 Gb/s links, respectively.

Data Storage

The Grid'5000 architecture provides two levels of data storage:

Local disk. Each compute node is equipped with a local hard disk accessible for user applications. It is however reserved for temporary storage, all data being deleted when the user job is completed.

Shared storage. A network file system (NFS) [52] is deployed on each site, the shared storage server being available from each compute node. It enables users to persistently store data on Grid'5000. This feature is essential for the deployment of distributed applications: the application code and libraries can be installed in the shared storage space and then the user can directly access data and execute the code on multiple compute nodes. Nevertheless, the NFS servers are neither replicated, nor synchronized between Grid'5000 sites. These operations must be manually performed by the user, if a multi-site deployment is required.

8.1.2 Grid'5000 experimental tools

Grid'5000 provides a series of tools that enable users to carry out large-scale experiments and facilitate the deployment of customized environments:

OAR [5] is a batch scheduler that allows Grid'5000 users to issue fine-grained reservations, ranging from one processor core to Grid-level reservations spanning over several sites and hundreds of nodes. Users can submit batch jobs, but also create advance reservations or interactive jobs. Aside from the job scheduler, OAR also provides a set of tools for real-time monitoring of the platform status and node availability.

Kadeploy [4] enables users to create and deploy customized operating system images on the Grid'5000 infrastructure. Users benefit from administrator rights on their environments, being thus able to install specific software required by their applications. Moreover, Kadeploy allows users to control the entire software stack and the reproducibility of their experiments.

The Grid'5000 API is a set of well-defined interfaces that enable secure and scalable access to resources in Grid'5000 from any machine through standard HTTP operations.

Taktuk [18] is a tool designed for efficiently managing parallel remote executions on large scale, heterogeneous infrastructures. It is a configurable and versatile tool that can be used as a fast and scalable deployment solution for distributed applications.

8.2 Automatic deployment tools

To perform complex experiments involving hundreds of nodes and various interconnected distributed systems, we developed a deployment framework for the Grid'5000 environment. It consists of a set of configurable scripts that enable an automatic and fine-tuned deployment and execution of large-scale experiments.

The goal of the deployment framework is to be an automatic and reliable tool that can assist users in performing large-scale evaluations. To achieve this goal, it has to match to following properties:

Configurable components. The self-adaptation frameworks we developed for BlobSeer consist of heterogeneous entities that have different requirements in terms of configuration. A suitable deployment tool should deal with this prerequisite and it should enable users to add and configure new components to a specific deployment in a straightforward manner.

Automatic execution. Each experiment may involve a large number of independent evaluations, such as measuring the performance of data transfers when the number of storage nodes is increasing. To this end, it is essential to have a tool that allows users to specify a set of parameters that can vary across the evaluation and to let the framework take care of the repeating the test with the successive sets of parameters.

Repeatable experiments. To improve the accuracy of the obtained results, the experiments have to be repeated several times under the same conditions. Our framework has to re-execute each experiment automatically, relying on a number of retries fixed by the user.

Flexible execution. The framework should allow users to develop their experiments in a flexible manner, so that they can perform evaluations that do not depend on predefined parameters or system settings.

We built a deployment tool to perform experiments that involve the self-adaptation components introduced in the previous chapter. The deployment framework comprises several key elements, designed to address the previous requirements:

Global settings. This component allows the user to define the systems that have to be included in the deployment, along with a set of global parameters required by the employed infrastructure and middleware. For instance, to use the Grid'5000 platform, one has to specify the type of OAR reservation used and the required identifiers, such as the reservation id and associated key. Moreover, it includes the type of connector used to contact the nodes (e.g. ssh), a list of the Grid'5000 sites involved in the deployment and global environment variables. The framework is in charge of propagating the global variables on all the nodes and of synchronizing the configurations on each Grid'5000 site.

Component specification. Each system involved in a deployment needs a specific set of scripts describing its configuration and deployment requirements. Such scripts can be easily integrated into the deployment framework by following a predefined format. Moreover, the description of each component comprises a configuration file that contains specific settings and requirements. As an example, the BlobSeer description includes parameters for the number of *data* and *metadata providers*, whereas the *Introspection Repository* configuration file specifies the number of servers and the details of the database.

Experiment design. The framework is designed to enable flexible testing relying on configurable evaluation parameters. The user can define *initialization scripts*, to inject specific input parameters into the deployed components, and *execution scripts*, to describe the actual test. Additionally, the user can specify various parameters to be used during the execution and a list of values for each of them. The framework will then automatically repeat the experiments using all the possible combinations of the given input values.

Results handling. To collect all the logs generated by the deployed platforms, our framework takes advantage of the shared user directory enabled by the NFS on Grid'5000. Moreover, the executed applications can generate their own result files and logs. The user can specify the relevant output files and the framework is in charge of gathering and archiving them in a single file for each experiment.

8.3 The introspection architecture

We evaluated the feasibility of gathering and interpreting BlobSeer-specific data by testing the *Introspection Layer* built on top of it. We employed the *Introspection Framework* to collect raw data from BlobSeer, process it and extract significant information regarding the state and the behavior of the system. To this end, we performed a series of experiments that assess the intrusiveness of the *Introspection Framework* and illustrate some of the collected data by means of a visualization tool.

8.3.1 Impact on the Blobseer data-access performance

This experiment is designed to evaluate the impact of using the BlobSeer system in conjunction with the introspection architecture. The *Introspection Layer* collects data from BlobSeer without disrupting the interactions between its components, and thus no constraint is enforced on the user's accesses to the BlobSeer entities. In this way, the throughput of the BlobSeer system is not influenced by any of the self-* frameworks. The only downside of such a system is the intrusiveness of the *Instrumentation Layer* that runs at the level of the BlobSeer components and may decrease their performance.

For this experiment we used the Grid'5000 clusters located in Rennes and Orsay. The nodes are equipped with x86_64 CPUs and at least 2 GB of RAM. We used a typical configuration for the BlobSeer system, consisting of 150 *data providers*, 20 *metadata providers*, one *provider manager* and one *version manager*. Both *data* and *metadata providers* store data on their hard disks and they are configured to store up to 64 GB and 8 GB, respectively. The MonALISA monitoring services are deployed on 20 nodes and they collect monitoring data from all the *providers*, each of them being dynamically assigned to a monitoring service in the deployment phase. To persistently store all monitored parameters, we employed an *Introspection Repository*, which was deployed on a dedicated physical machine within Grid'5000.

This test consists of deploying a number of concurrent clients that make a single WRITE operation. Each client writes 1 GB of data in a separate BLOB, using a chunk size of 8 MB. We analyze the aggregated throughput of the BlobSeer WRITE operation obtained when deploying it standalone compared to the BlobSeer outfitted with the *Introspection Layers*. The throughput is measured for a number of clients ranging from 5 to 80 and the experiment was repeated 3 times for each value of the number of deployed clients.

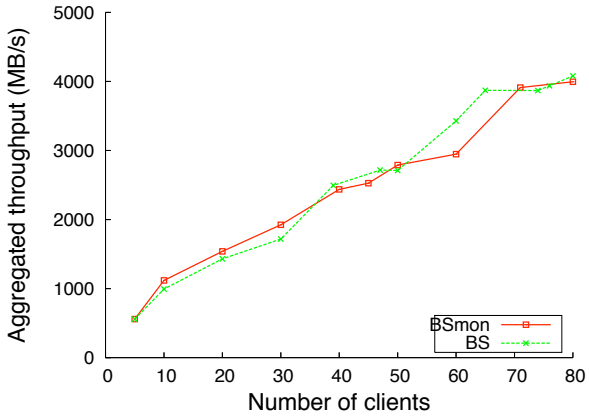


Figure 8.1: The aggregated throughput of the WRITE operation for BlobSeer (BS) and for BlobSeer with the monitoring support enabled (BSMON).

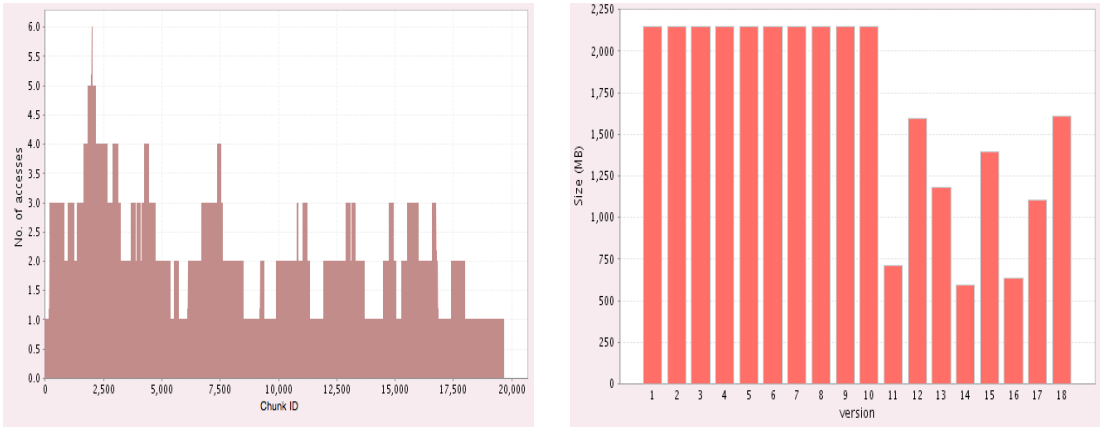
Figure 8.1 shows that the performance of the BlobSeer system is not influenced by the addition of the instrumentation code and the generation of the monitoring parameters, as in both cases the system is able to sustain the same throughput. Moreover, the test evaluates the performance of the BlobSeer system as the number of generated monitoring events increases: since the *Introspection Layer* computes its output based on the monitored data generated for each written chunk, the more fine-grained BLOBS we use, the more monitoring information has to be processed. For this test, each BLOB consists of 128 chunks and therefore the data-access performance is preserved even when the number of generated monitoring parameters reaches 10,000, as it is the case when testing with more than 80 clients.

8.3.2 Visualization tool for BlobSeer-specific data

To provide a graphical representation of the most important parameters yielded by the *Introspection Layer*, we implemented a visualization tool on top of a MonALISA monitoring repository. Its goal is to allow administrators to visualize node utilization, data-access trends or BLOB statistics by using an intuitive graphical interface.

We show the outcome of the introspection layer through an evaluation performed on 127 nodes belonging to a Grid'5000 cluster in Rennes. The nodes are equipped with x86_64 CPUs and at least 4 GB of RAM. We deployed each BlobSeer entity on a dedicated node, as follows: two nodes were used for the *version manager* and the *provider manager*, 10 nodes for the *metadata providers*, 100 nodes for the *storage providers* and 10 nodes acted as BlobSeer clients, writing data to the BlobSeer system. Four nodes hosted MonALISA monitoring services, which transferred the data generated by the *Instrumentation Layer* built on top of the BlobSeer nodes to the MonALISA repository. The repository is the location where the data were stored and made available to the visualization tool.

In this experiment, we used 10 BLOBS, each of them having a chunk size of 1 MB and a total size larger than 20 GB. We created the BLOBS and we wrote 10 data blocks of 2 GB on each BLOB. Each data block overlaps the previous one by 10%. Next, we started 10 clients in parallel and each of them performed a number of WRITE operations on a randomly selected BLOB. The blocks were written on the BLOB at random offsets and they consisted of a



(a) Number of WRITE accesses on each chunk of a BLOB, (each chunk is identified by its position within the BLOB).

(b) The size of all the stored versions of a BLOB.

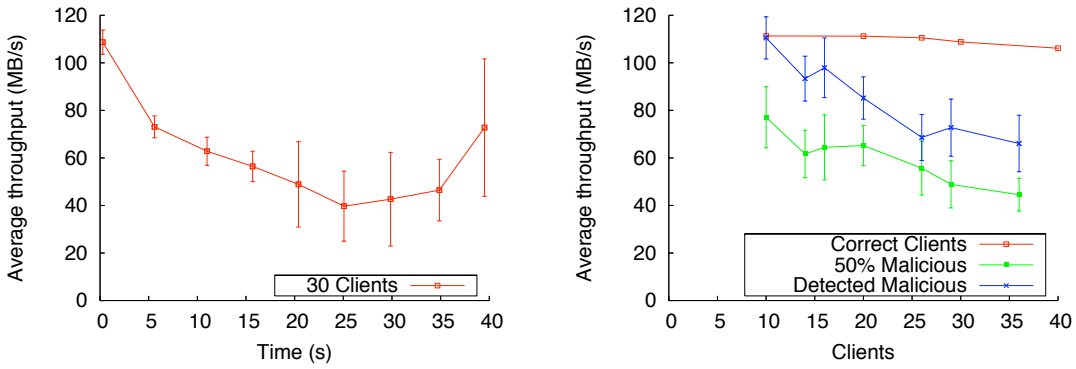
Figure 8.2: Visualization for BlobSeer-specific data.

random number of chunks, ranging between 512 MB and 2 GB in size.

We processed the raw data collected by the *Monitoring Layer* and extracted the higher-level data within the *Introspection Layer*. Some results are presented below, along with their graphical representations.

Access patterns. They represent a significant information that the introspection layer has to be aware of. This information can be obtained by computing the number of READ/WRITE accesses. The access patterns can be examined from two points of view. The first one regards the access patterns for each BLOB. It considers the number of READ or WRITE accesses for each chunk, for a specified version or for the whole BLOB, and it identifies the regions of the BLOB composed of chunks with the same number of accesses (Figure 8.2(a)). On the other hand, the access pattern visualization may focus on the number of READ or WRITE operations performed on each provider, allowing for a classification of the providers according to the pressure of the concurrent accesses they have to withstand.

The size of all the stored versions of a BLOB. The differences between the successive versions of the same BLOB are presented in Figure 8.2(b), where the size of the new data introduced by each version into the system is shown in MB. This information, correlated with the number of accesses for each version, can be used to identify versions that correspond to a small amount of data and are seldom accessed. The number of versions and the number of accesses for each of them may as well be necessary for a component that handles an automatic replication mechanism for intensively-used BLOBS or versions.



(a) The evolution of the average throughput when 15 clients out of 30 perform malicious writes.

(b) The average throughput under correct and malicious writes.

Figure 8.3: Impact of malicious users on data-access performance.

8.4 The security framework

We evaluated the impact of enforcing security policies on top of the BlobSeer system and the performance of the *Policy Management* module, through a series of large-scale experiments performed on Grid'5000.

8.4.1 Experimental setup

We used the clusters located in Rennes and Orsay, which include compute nodes equipped with x86_64 CPUs and at least 2 GB of RAM.

For all the experiments, we employed the same deployment settings for the BlobSeer system. We used a typical configuration that enables the system to store massive amounts of data that can reach the order of TB. It consists of 50 *data providers*, 15 *metadata providers*, one *provider manager* and one *version manager*. Both *data* and *metadata providers* are configured to store data in memory, and to persistently save it on the disk in a background thread. In addition, we used 8 nodes for the monitoring services, which collect the user activity information. The *User Activity History* is stored on a dedicated node, which also hosts the *Policy Management* module. Each entity is deployed on a dedicated physical node.

Each experiment is composed of two phases. In its first phase, all BLOBS required by the experiment are created. Its second phase consists of WRITE operations executed concurrently by the users, each of them generating data in a separate BLOB.

We focused on the video surveillance scenario described in Section 6.2, in which a Cloud storage service is needed to store continuous flows of data recorded by the cameras. The video surveillance cameras are modeled as BlobSeer users that perform a sequence of WRITE operations. All users run concurrently and each of them performs 10 writes to BlobSeer, each written data block having a size of 256 MB. The typical chunk size employed in the BlobSeer system is 64 MB. Therefore, a correct user always uses this chunk size for its writes, as it guarantees a constant throughput sustained by the storage system. In this context, we define a DoS attack as a write operation in which the number of chunks written before publishing is much larger than the number of chunks generated by a correct user for the same size of

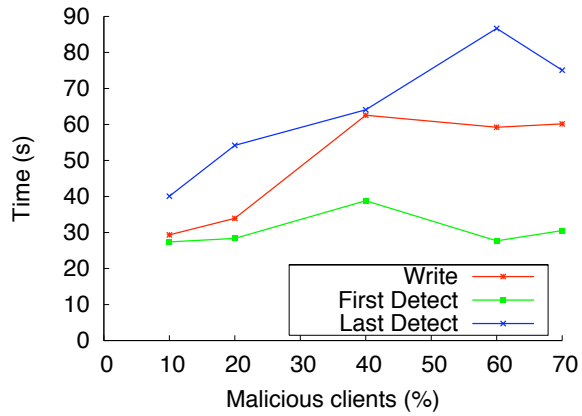


Figure 8.4: The write duration and the detection delay when 50 concurrent clients write to BlobSeer.

the write. As a consequence, we simulate the DoS attacks as malicious users that write the same amount of data, i.e. 256 MB, but use a much smaller chunk size: 2 MB.

8.4.2 Impact of malicious users on data-access performance

The first experiment shows the evolution in time of the average throughput of concurrent users that write to BlobSeer when the system is subject to DoS attacks. For this test we used 30 concurrent users, each of them performing 10 writes. Half of the users have a malicious behavior, performing DoS attacks. To study the impact of our security framework, we defined a security policy that sets a limit on the number of chunks a user can write before publishing the full write and we enabled the *Policy Management* module for the experiment. Figure 8.3(a) shows that the initial average throughput has a sudden decrease when the malicious users start attacking the system. As the *Policy Management* module detects the policy violations, it feeds back this information to BlobSeer, enabling it to block the malicious users, when they issue requests for more data providers to write chunks on. As a consequence, the average throughput for the remaining users increases back towards its initial value.

The goal of our second experiment is to assess the impact of concurrent DoS attacks on the performance of the storage system. Figure 8.3(b) shows the average throughput of concurrent users that write to BlobSeer, when the number of users ranges from 10 to 40. The results correspond to three different scenarios: (1) all the users perform correct writes, (2) 50% of the users have a malicious behavior and no security mechanism is enabled to protect the system and (3) 50% of the users have a malicious behavior and the *Policy Management* module is enabled. When all the concurrent writers act as correct users, the system is able to maintain a constant average throughput for each of them. However, when no security mechanism is employed and half of the users attempt a DoS attack, the performance is drastically lowered for every user that accesses the system. Further, the results demonstrate that the throughput increases again, once the attackers are blocked by the *Policy Management* framework.

8.4.3 Performance evaluation of the Security Management Framework

In order to efficiently protect BlobSeer against security threats, the *Policy Management* module has to expose attacks as fast as possible, so as to limit the damage inflicted to the system and to minimize the impact on the correct users.

We measured the detection delay when the percentage of malicious users increases from 10% to 70% out of a total of 50 users. For each percentage of malicious users, Figure 8.4 displays the duration of the writes performed by all the users (a sequence of 10 write operations of 256 MB each) and the delays between the beginning of the write operation and the moments when the first and the last malicious users are detected, respectively. The results show that the time needed to detect and block the malicious users is comparable to the time it takes to write the data into the system. We measured an average delay of 20 seconds between the total writing time and the time needed to detect all the malicious users. It shows that the *Security Framework* is able to promptly react when an attack is initiated, to enable legitimate operations to proceed by blocking malicious data transfers and to restore data-access performance once the attackers are eliminated.

Part III

Integrating and evaluating BlobSeer in Cloud environments

Chapter 9

The Nimbus cloud environment

Contents

This chapter focuses on the Nimbus Cloud project, giving an insight on its design and on the features it provides to the scientific community. Furthermore, we discuss the implementation of the Nimbus IaaS Cloud and we detail the Cumulus storage service, which plays the role of the Cloud data service in Nimbus.

9.1 The Nimbus Cloud infrastructure

Nimbus is an open-source Infrastructure-as-a-Service Cloud platform, designed to address the needs of the scientific community in multiple contexts, ranging from high-energy physics [50] to bioinformatics [66]. It provides on-demand virtualized computational resources that the clients can fully customize and have complete control over the applications that are executed on them.

9.1.1 Architecture details

The architecture of a Nimbus Cloud is based on four modular components, which deliver the basic Infrastructure-as-a-Service functionalities to the users. The Nimbus core is devised as an extensible framework, on top of which additional modules were built to enable easy virtual cluster configuration, interfaces to other IaaS Clouds or optimized virtual machine scheduling on physical resources.

The main components of a Nimbus Cloud are detailed below.

The Cloud Client. It implements a wide range of commands to facilitate user's access to Cloud services. The *Cloud Client* provides powerful tools for provisioning and managing virtual resources. Additionally, it automatically handles the creation of configuration files, user authentication, and hides the calls and file transfers required to perform complex environment deployments.

The Workspace Service. Placed at the core of the Nimbus architecture, the *Workspace Service* is a standalone site VM manager that plays the role of the Cloud entry point. It coordinates virtual machines life cycle and manages the compute nodes that make up the Cloud. To interact with Cloud users, the *Workspace Service* implements a set of access protocols that they can invoke to perform virtual environment management operations. Furthermore, it incorporates user authentication and authorization mechanisms developed within the Globus [33] framework Grid Security Infrastructure (GSI) [2].

The Workspace Control. Designed as an portable agent running on each node, this component is in charge of VM deployment, management and configuration at the level of each Virtual Machine Manager (VMM) node. It is built to work with various hypervisors, such as Xen [10] or KVM [60]. Its tasks include monitoring locally-deployed VMs and interacting with the *Workspace Service*.

Cumulus. It provides Cloud data services for the Nimbus platform, playing the role of the Nimbus storage repository for VM images. It features an open-source implementation of the S3 REST API that enables simple data management through existing tools developed for Amazon S3. Cumulus provides users with a VM image storage repository accessible through standard interfaces, thus being integrated into Nimbus as an essential tool in VM management and deployment.

9.1.2 Infrastructure-level services

To enhance the range of services provided by the Nimbus cloud, several additional components have been designed to address specific problems in Infrastructure-as-a-Service Cloud environments:

LANTorrent. This file distribution protocol provides VM image propagation services for the Nimbus IaaS Cloud. It is built on top of a multi-cast protocol optimized for large file transfers from a central location to a set of destination nodes. *LANTorrent* is embedded into the VM management layer, being the main VM image propagation mechanism employed by the Nimbus toolkit.

Resource manager. The elastic site manager [63] is a component devised to dynamically incorporate physical clusters into a Nimbus Cloud. It can obtain access to new nodes by interacting with local resource managers, such as Torque, so as to collect information about the site usage and the available nodes. Furthermore, it can provision nodes from other Nimbus-based Clouds or from public Clouds, like Amazon's EC2.

Backfill VMs. In the context of IaaS clouds, the backfill VMs are defined as generic VMs provisioned automatically by the IaaS framework using preemptible leases [64]. Such VMs are suitable for High-Throughput workloads (HTC), typically consisting in jobs that perform independent computations and do not need to synchronize. The backfill VMs mechanism ensures such jobs submitted by local schedulers (e.g., Condor [103]) will eventually execute, although it can terminate VMs at any moment to accommodate standard, on-demand workloads.

WSRF/EC2 frontends. The WSRF service implements the default protocol employed by the *Workspace Client*. The *EC2 frontend* provides support for the SOAP and Query interfaces supplied by Amazon's EC2. They allow users to easily import their applications from EC2 Clouds to Nimbus, by enabling standard EC2 client implementations to work seamlessly against Nimbus Clouds.

Workspace Pilot. This component was developed to enable site local resource managers (LRM) to run jobs on the idle physical nodes that make up the Cloud. It can interact with the *Workspace service* and shut down existing jobs when the Cloud manager decides to use the nodes for hosting VMs.

9.1.3 Platform-level services

Recently, the Nimbus project has introduced a set open-source tools that extend the initial Infrastructure-as-a-Service functionality of Nimbus to higher-level services described as follows:

Cloudinit.d. This is a tool designed to facilitate large-scale application deployment, management and configuration. It includes mechanisms for provisioning virtual machines over a set of IaaS Cloud providers, either commercial or open source, on top of which the applications are then installed, customized and executed. The main features of the *Cloudinit.d* tool include easy and repeatable VM clusters deployment, coordination for interdependent application launches, support for federated Cloud provider and for application monitoring.

Context Broker. This component enables users to deploy "one-click" virtual clusters in a fast and configurable manner. It is responsible for deploying a set of virtual machine images specified by the user, and for customizing them according to user-defined roles. Such roles can define authentication rules for individual or groups of VMs, as well as runtime-generated configuration files and application launches.

9.1.4 Virtual Machine lifecycle

The lifecycle of a Nimbus virtual machine follows the typical pattern defined by many Infrastructure-as-a-Service Clouds, as detailed below.

Image creation. To execute an application in a Cloud environment, a user must first register a virtual machine image with the Cloud middleware. Many commercial Clouds provide standard images, together with a description of their features and the associated price. Moreover, research communities have also created and made public various virtual machine images specifically tuned for scientific applications [20]. Once the user has access to such an image, it has to upload it into the Cloud system. In Nimbus, this can be done by uploading the image into Cumulus, through the Nimbus Client tools, or by using S3 standard tools.

Launching VMs. Nimbus provides two mechanisms for running virtual machines. The first one implies selecting a VM image and launch it by employing the *Cloud Client* command line tools. More complex deployments that involve multiple VMs deployed simultaneously can be performed through the *Context Broker*. In this case, the user has to create a configuration file describing the requirements of its virtual cluster, such as the number of VMs or the VM images used to launch one or several VMs. More advanced settings can be handled by the *Context Broker*, including network configuration or user access policies. The *Cloud Client* handles the cluster creation. Upon booting, each VM then contacts the *Context Broker* to perform the contextualization tasks.

Stopping VMs. A launched VM can be manually stopped by the user when its applications have completed their execution. Additionally, the Cloud service itself may shut down a running VM, if the lease it obtained upon launch has expired.

Saving a modified VM. A user can install specific applications on a running VM, as it has full control over the operating system. To be able to subsequently use the modified VM, the user can save it back into the Cloud repository. The *Cloud Client* provides commands to snapshot a running VM.

9.2 The Cumulus storage system

Cumulus is an open-source Cloud storage system that combines efficient data-transfer mechanisms and data-management techniques, aiming at providing data Cloud services in the context of scientific applications. It is designed to support swift data transfers using S3-compatible interfaces. Its features include simple upload/download operations, fair-sharing among concurrent users and quota support [14].

9.2.1 The architecture of Cumulus

Cumulus is designed as a modular system that features a set of interface layers to enable a straightforward and efficient interaction with external modules. Its architecture is detailed in Figure 9.1.

Service Interface Layer. This module is responsible for recording and interpreting client commands. It implements Amazon's S3 REST protocol, which is the most widely used protocol for commercial Clouds. Thus, the service can process client requests initiated through various libraries developed for S3, such as boto [104] or s3cmd [92].

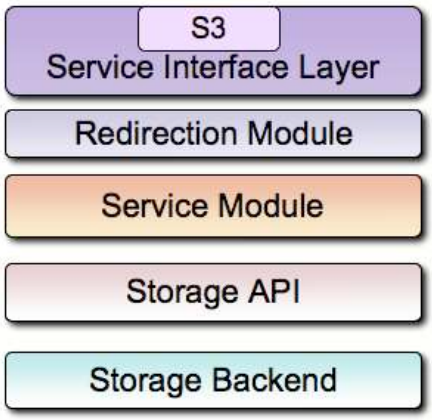


Figure 9.1: Architecture of the Cumulus service.

Redirection Module. To enable Cumulus to efficiently manage concurrent clients, a *redirection module* provides support for replicated Cumulus servers backed by the same storage service. It acts as a load balancer, forwarding the client connections to another server if its workload exceeds some predefined limits.

Service Module. This module represents the core of the Cumulus system. Its role is to translate client requests into calls to the storage backend and to send back the responses, along with the eventual error messages. It includes an authentication module backed by a database that stores the file and bucket permissions as set by the clients that created them. Each client request is first validated by the authentication module, which verifies the client's rights with respect to the stored access control list (ACL) of the requested object. Once the request obtains the necessary authorization, the service invokes the appropriate method on the *Storage API* to process it.

Storage API. This module provides an interface that allows system administrators to customize their service according to the Clouds requirements. Thus, using a Cumulus service on top of a local disk may be enough for a Cloud repository serving small deployments of only a few virtual machine images. Employing Cumulus as a storage service for large-scale applications with many concurrent clients may require, however, a distributed or parallel file system as a storage backend. This module separates the processing of client requests from the actual implementation of the storage backend, making Cumulus a versatile tool that can be adapted to various contexts.

Storage Backend. The default storage backend shipped with the Cumulus service implements the interaction with a POSIX-compliant file system. It provides support for interconnecting Cumulus with either local file systems or parallel file systems that expose a POSIX interface, such as NFS, PVFS or GPFS.

9.2.2 Main features

The essential aspects that make Cumulus a suitable framework for various Cloud storage tasks are detailed below.

S3-compatible interface. Amazon's S3 REST protocol is the "de facto" standard for data services in Infrastructure-as-a-Service Clouds. As Cumulus implements this protocol, it is thus compatible with a wide range of tools designed for Amazon S3. The potential clients can simply switch to Nimbus, without any change in their VM management tools.

Eventual consistency. To provide a service which is fully-compatible with Amazon's S3 implementation, Cumulus provides the same consistency guarantees as S3. More specifically, it implements an eventual consistency model for overwriting PUT operations and read-after-write for new object PUT calls.

Configurable storage backend. This is a crucial design choice that shapes Cumulus into a flexible data-management system able to adapt to various configurations. According to the needed level of storage reliability and performance, the system administrators can easily implement new storage backends to match their specific requirements.

Support for replicated servers. When a large number of concurrent clients attempt to access the stored data, the system may face significant bursts of requests, which can decrease its performance. To efficiently handle such circumstances, which are representative in the context of data-intensive applications, Cumulus employs a load-balancing mechanism to split the load among various servers backed by the same storage layer.

Scalability. Cumulus is able to take advantage of multiple servers running in parallel, thus offering improved performance and optimized data transfers for concurrent clients.

Fair sharing among clients. Experiments performed in [14] for a large number of simultaneous data accesses show that Cumulus is able to sustain a constant average transfer throughput for each client. Fair sharing is essential for scientific applications relying on distributed workers that concurrently generate or read data.

9.3 Summary

Nimbus is a popular open-source Infrastructure-as-a-Service Cloud toolkit specifically designed to address the needs of the scientific community. Its usage focuses on scientific applications, which drive as well the development of additional Nimbus components that optimize their performance, deployment and manageability in a Cloud environment. We chose to conduct our Cloud-related evaluations on top of Nimbus, as it is a state-of-the art IaaS Cloud providing a wide-range of features, as well as a flexible and modular architecture.

We target data-intensive workloads in Cloud environments. To investigate data management challenges under such scenarios, we focused on Cumulus to provide us with an S3-compatible, Cloud-oriented framework. Among the existing open-source Cloud data services, Cumulus offers an improved flexibility with respect to the capabilities of the storage backend. It also provides an appropriate framework for assessing the needs of Cloud data services and for evaluating existing storage solutions against Cloud scenarios.

Chapter 10

A BlobSeer-based backend for Cumulus

Contents

This chapter introduces our contribution with respect to Cloud data storage: we designed and implemented a BlobSeer-based distributed backend for the Cumulus service. Its goal is to provide high-throughput data transfers for Cloud environments, as well as to optimize Cumulus for highly-concurrent accesses to data. To this end, we enhanced BlobSeer with a file-system layer, enabling it to meet the requirements of the interface defined by Cumulus. Furthermore, we briefly discuss specific implementation details and we provide an evaluation of the storage service through synthetic benchmarks.

10.1 Towards a file-system interface for BlobSeer

We designed a file-system layer on top of BlobSeer to enable a straightforward implementation of file-oriented interfaces backed by BlobSeer. We aim at providing an easily-accessible file-system interface and a hierarchical file namespace, while preserving the efficient concurrent data operations built into BlobSeer.

The reasons for using BlobSeer in this context are detailed in the following section, which summarizes the main requirements for data-management systems in Cloud environments. Next, we introduce the file-system API we implemented on top of BlobSeer and we give some insights on the design of the BlobSeer file system.

10.1.1 Requirements for the storage backend

Data-intensive paradigms, such as MapReduce, have recently gained a considerable interest from the Cloud computing community. As the data processed by such applications is increasing exponentially, data-management solutions have become a crucial aspect that impacts the adoption rate of the Cloud. Consequently, data Cloud services have to comply with several requirements specific for large-scale data-intensive applications.

Massive files. Data-intensive applications typically process huge amounts of records, which cannot be hosted in separate, small files. To efficiently store such datasets, data services have to collect all the records into massive files that can reach Terabytes of data in size.

Fine-grained access. Large datasets usually comprise billions of small records. Distributed processing paradigms split such datasets among computing nodes, which in turn are responsible for accessing and processing specific chunks of data. The performance of a data-intensive application is thus dependent on the backend storage system, which has to provide efficient access to small blocks of the same dataset for multiple concurrent processes.

High-throughput concurrent data transfers. Parallel data processing is one of the crucial features that allow data-intensive applications to accommodate large amounts of data. To enable efficient support for such applications in the Cloud, data-management platforms have to sustain high-throughput data transfers, even under heavy access concurrency.

Fault tolerance. Reliability is a key requirement for Cloud storage, especially in public Cloud environments. It can be achieved by employing fault-tolerant storage backends. Moreover, *data versioning* is a desirable feature in such contexts, as it enables a transparent support for rolling back incorrect or malicious data modifications.

The aforementioned requirements represent the design principles of BlobSeer, a data-management system specifically designed to address the challenges of data-intensive applications. Its efficient support for massive data storage, as well as its ability to sustain high-throughput data transfers under heavy client concurrency, account for our approach of using BlobSeer as a Cloud data storage backend.

10.1.2 The BlobSeer Namespace Manager

To equip BlobSeer with a file-system interface, we designed a *namespace manager*, a centralized entity in charge of the following operations.

Manage the file hierarchy. We introduced a hierarchical directory structure on top of BlobSeer, where each file is mapped onto a BLOB and directories are only managed by the *namespace manager*. The *namespace manager* also stores the file system metadata.

Map files to BLOBS. Each file name in the hierarchy is backed by a BLOB, while the directories only contain metadata. This approach allows the file-system layer to interact with BlobSeer only to perform data-access operations on files. Moreover, we provide versioning support for files, as they are equivalent with standard BLOBS.

Implement the file system API. The *namespace manager* introduces a file system interface for BlobSeer. It implements the typical file-system operations, ranging from directory management to file access.

10.1.3 The file system API

The BlobSeer file system provides a specific API that exposes standard file-system operations, such as create/list directories, create/open/read/write/close files. However, the file system layer does not implement the POSIX semantics, preserving instead the definition of the original Blobseer primitives. This design choice has several consequences on the file system API:

Versioning interface. The file system exposes file versions in the same way as BlobSeer, providing specific versioning-oriented primitives. The implementation of the *namespace manager* only manages file names, relying on BlobSeer to retrieve the information related to the versions. Thus, the file-system layer enables clients to directly access the underlying BlobSeer primitives and does not impact on the efficient versioning support in BlobSeer.

Consistent concurrent writes. BlobSeer is optimized to provide support for multiple concurrent clients that write to the same BLOB. Its versioning-based design guarantees the atomic creation of a new version each time the BLOB is modified. We preserve these features at the file-system level, exposing the specific BlobSeer consistency semantics for the WRITE operation. This is a key aspect allowing the BlobSeer file system to obtain high throughput data transfers under heavy concurrency.

Support for concurrent appends. BlobSeer introduces APPEND support as an alternative for writing to a specific location of the BLOB. Many distributed applications that process small records, such as web pages, may create billions of KB-sized output files. Managing the file-system metadata associated with such a huge amount of small files can be a very time-consuming task impacting the performance of the storage system. To avoid this namespace-management overhead, we also provide the APPEND primitive at the level of the file system. to enable applications to collect the results generated by concurrent processes into the same output file.

10.1.4 Introducing a 2-phase write operation

Apart from the basic file-system operations, we introduced two new primitives to optimize the performance of writing data to BlobSeer through higher-level services, such as Cumulus. The Cumulus transfer protocol streams KB-sized chunks of data from the user to the underlying storage system. Uploading a file into the storage system requires the following steps.

Open file. The backend storage service has to provide a file handle for the data-transfer operations.

Write data chunks. As data is streamed from the client, Cumulus caches small data chunks that are sent to the storage backend through a series of WRITE operations.

Close file. At the end of the upload primitive, the Cumulus service closes the file and discards the provided handler.

Each WRITE operation performed by the BlobSeer client library results in new metadata and a new BLOB version, as detailed in Chapter 5. Consequently, translating a Cumulus upload into a series of BlobSeer WRITE calls would be equivalent to creating a large number of versions for a single uploaded file. Such an approach would incur several disadvantages. First, generating new versions for each data chunk accounts for an increased pressure on the *metadata providers* and on the *version manager*. As a result, an upload operation can be subject to an important performance penalty and also impact the throughput of other concurrent transfers, by creating contention at the level of the *metadata providers* and *version manager*. Another drawback is that only the last version of the set generated by the upload would be valid, the others reflecting only an incomplete view of the uploaded file. The system should thus provide some mechanisms to label the valid version.

To avoid the aforementioned limitations, we introduced a *2-phase write operation* in the BlobSeer client library. It basically consists in splitting the regular BlobSeer WRITE into the following primitives, which are also exposed at the level of the file-system interface:

Chunks write. This operation involves requesting a set of *data providers* from the *provider manager*, and writing a range of chunks to those *providers*. The primitive returns the list of chunk identifiers and the *data providers* storing them as an output, so as to enable the client to keep track of the successfully written chunks.

Write publication. To complete a BlobSeer WRITE operation, the *write publication* primitive has to be called with the write information issued by the *chunks write* as a parameter. This operation is responsible for building the metadata tree associated with the written chunks and for publishing the new version to the *version manager*. Note that the written chunks must make up a continuous range in order for the write to be valid as a new BlobSeer version.

The 2-phase write does not impact the performance of concurrent writes in BlobSeer, as it enables chunk writes to be performed in parallel just as they were carried out in BlobSeer. Furthermore, the write publication step does not change the Blobseer versioning mechanism, nor the semantics of the complete WRITE.

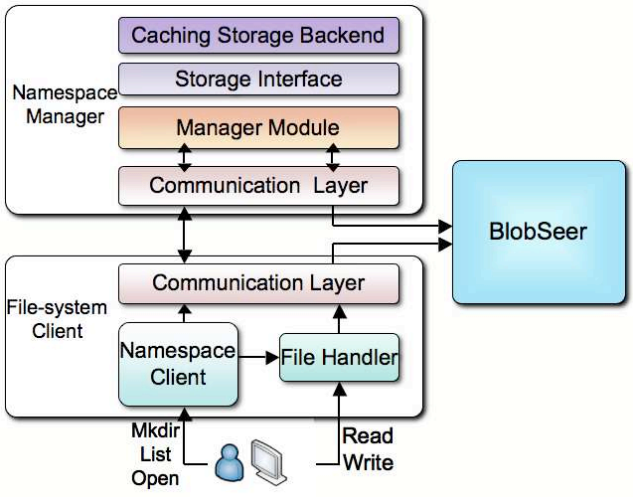


Figure 10.1: Implementation details of the BlobSeer file system.

10.2 Implementation

This section describes the implementation of the BlobSeer file-system layer and details the key features of the BlobSeer-based backend for Cumulus.

10.2.1 Designing the BlobSeer file system

The implementation of the BlobSeer file system relies on two main components depicted in Figure 10.1. The *namespace manager* is designed as a standalone server responsible for storing the file-system hierarchy. The other key component is the *file-system client*, implemented as a library that exposes the BlobSeer file-system API. Both components are detailed as follows.

The Namespace Manager

The goal of the *namespace manager* is to manage and store the file system namespace and the file metadata. It is implemented as a C++ server that can asynchronously serve concurrent client requests. As Figure 10.1 shows, it features four main modules:

Manager. This is the key module in charge of processing client requests related to file-system metadata operations. The *namespace manager* benefits from a lock-free, efficient implementation. This is due to the underlying *Communication Layer*, which is designed to serialize the incoming requests into a request queue and to forward one at a time to the upper layers.

Communication Layer. The communication between the *namespace manager* and the client relies on the RPC layer designed for BlobSeer and detailed in [75].

Storage Interface. We designed a extensible storage interface that allows for various implementations of the namespace storage backend.

Caching Storage Backend. We built an in-memory storage backend to optimize namespace operations retrieval. It consists in a key-value pair map indexed by key. Each file or directory in the file system is uniquely identified by an absolute path, which is stored as a key by the *namespace manager*. For each key, the stored information includes the creation date, the type (i.e., file or directory) and the corresponding BLOB identifier for files. The namespace storage backend can be easily extended to persistently save the contents of the cache in a distributed fashion, by leveraging the BlobSeer *metadata providers* implementation. To this end, a hash function can be used to associate a *metadata provider* to a key, in the same way the BlobSeer client selects *metadata providers* to store BLOB metadata (this mechanism is detailed in Chapter 5).

The *namespace manager* handles two main request categories. First, the main function of the *namespace manager* is to maintain the file-system hierarchy and to enable the clients to modify it. To this end, it exposes a set of RPC calls for creating, moving, copying, deleting and listing directories. The requests that concern the directory structure are directly processed by the *namespace manager*, as they do not require an interaction with BlobSeer. File creation and opening, however, fall into the second type of requests. Each *file create* operation entails a call to the CREATE primitive in BlobSeer. If this operation succeeds, the returned BLOB identifier is stored by the *namespace manager* within the metadata information of the file. For each subsequent *open* operation, the *namespace manager* forwards the BLOB identifier to the client library, which then employs it to access the BlobSeer system for READ/WRITE operations.

The File-System Client

The file-system client is implemented as a C++ library that exposes the file-oriented BlobSeer API. To enable the BlobSeer file system to be used in a wide range of application contexts, we also built library bindings in C, Java and Python.

The client library has to be installed on each node involved in the interaction with the BlobSeer file system, as it is the key building block of any BlobSeer-based storage backend. Its implementation relies on the following components, depicted in Figure 10.1.

Namespace Client. This module intermediates the communication between the application layer and BlobSeer. It forwards the namespace-related user requests to the *namespace manager* and returns the reply. This module was developed on top of the BlobSeer RPC layer, which handles the serialization and transfer of the requests and their corresponding replies.

File Handler. Each time a file is opened, the *namespace client* module retrieves the associated BLOB identifier from the *namespace manager*. Further, it builds a *File Handler* object that exposes the API for I/O operations and encapsulates a BlobSeer client to access the obtained BLOB identifier. All the subsequent operations executed on the *FileHandler* are directly performed on the BlobSeer entities, without any interaction with the *namespace manager*, thus enabling the BlobSeer file system to sustain the same performance level as the standard, BLOB-oriented interface. It does not provide POSIX semantics, exposing only a WRITE operation that is directly mapped on the BlobSeer WRITE defined in

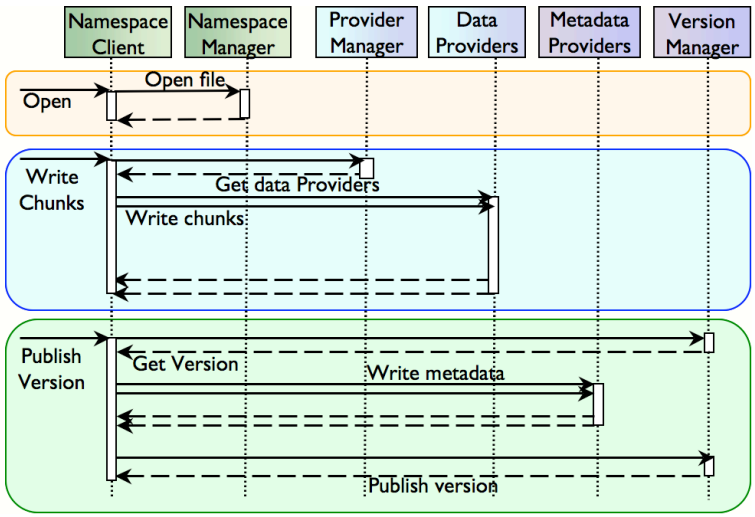


Figure 10.2: Implementation details of the BlobSeer file system.

Chapter 5. Additionally, the *File Handler* also provides the two primitives that make up the *2-phase write*, namely *Chunks write* and *Write publication*.

As an example, the diagram in Figure 10.2 illustrates the interactions between the client library, the file system layer and the BlobSeer entities in the case of the most complex file operation, the *2-phase write*. To perform I/O operations on a particular file, the client has to open the file and thus acquire the associated BLOB identifier. Next, the client initiates a set of *write chunks* calls on the received *File Handler*, which are translated into chunk transfers to the *data providers*. After all chunks are successfully written, the client can submit a *publication* request to complete the BlobSeer WRITE procedure.

To improve the I/O performance, the BlobSeer file system implements a client-side buffering mechanism, targeting both READ and WRITE operations. As each BlobSeer WRITE involves only whole data chunks, we buffer the small requests and submit the chunk to BlobSeer when it has been completely filled or when the client decides to publish the WRITE. The same technique is used for reading: the client library prefetches full chunks and serves the reads from its cache.

10.2.2 BlobSeer-based Cumulus backend

We implemented a Cumulus storage backend based on the BlobSeer file system, so as to make the essential BlobSeer features available in a Cloud environment. To this end, we implemented the two building blocks defined by the Cumulus Storage API.

Build a namespace-management class. This class defines the behavior of namespace-related operations, such as creating/deleting buckets, and opening or creating files. To interact with the BlobSeer file system, this class employs the *Namespace Client*, translating each operation into its counterpart in the BlobSeer client library.

Implement a data-access class. It is equivalent to a file handler class, which defines the operations related to the stored data. It is instantiated by the namespace management

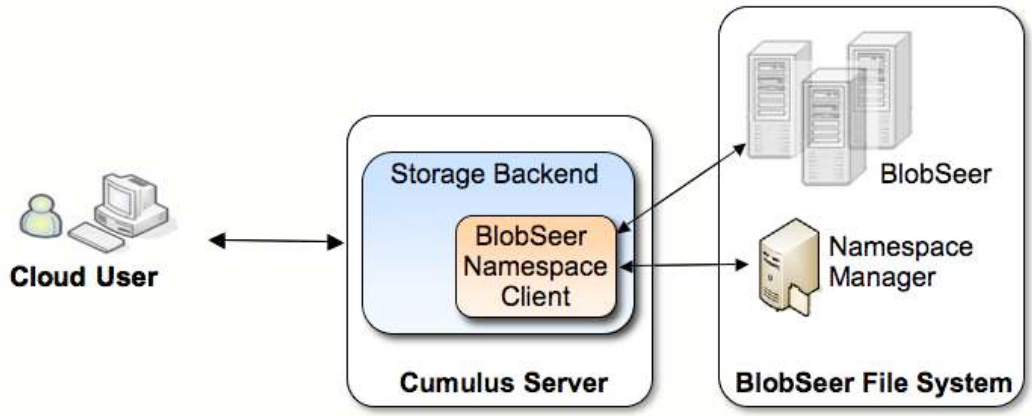


Figure 10.3: BlobSeer as a backend for the Cumulus service.

class whenever a client requires access to data and it facilitates data streaming to and from the storage backend. It is based on the BlobSeer *File Handler*, through which it relays the Cumulus requests towards the storage backend.

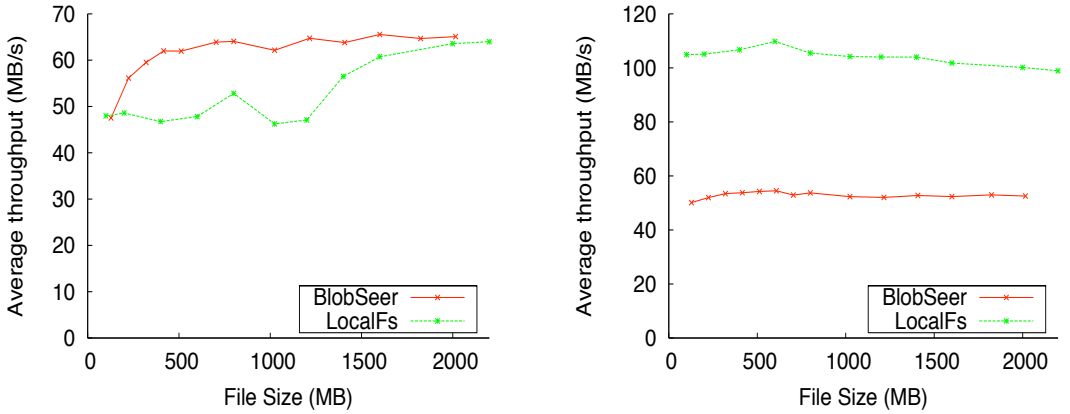
As shown in Figure 10.3, the Cumulus service acts as a BlobSeer client, which intermediates the data transfers between the S3-compatible users and the BlobSeer backend. Each Cumulus server is also able to perform concurrent transfers, thus starting multiple BlobSeer clients in parallel to interact with the backend. This property plays to BlobSeer's strenghts, as it is devised to support high-throughput concurrent connections.

A particular care was given to the implementation of the WRITE operation. This operation is performed on the backend each time a piece of data is streamed from the client. Typically, Cumulus streams small blocks of 4 KB. Since writing each KB-sized block to BlobSeer would generate an important performance overhead, we implemented the Cumulus WRITE on top of the *write chunks* primitive in the BlobSeer file system. This approach has two advantages. First, it benefits from the improved performance in terms of I/O throughput corresponding to the *2-phase write* in BlobSeer. Second, the Cumulus WRITE takes advantage of the client-side caching mechanism in the BlobSeer file system implementation, which collects all the contiguous small WRITES before issuing a WRITE into BlobSeer. The WRITE operation is fully completed only when the streaming process ends successfully and the Cumulus client calls the CLOSE procedure. Closing a file implies flushing all the cached data chunks and publishing all the written chunks as a new BLOB version.

10.3 Microbenchmarks

10.3.1 Environmental setup

We carried out a set of experiments on the Rennes cluster of the Grid'5000 platform. The used nodes are interconnected through a 1 Gbps Ethernet network, each node being equipped with at least 4 GB of memory. For each experiment, the BlobSeer deployment consists of one *version manager*, one *provider manager*, one node for the *namespace manager*. The number of *data* and *metadata providers* varies across experiments and it is specified for each of them. The



(a) Upload throughput for one client.

(b) Download throughput for one client.

Figure 10.4: Impact of the file size on the file-transfer throughput in Cumulus.

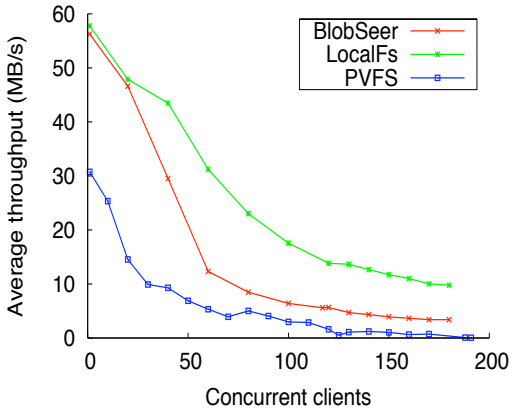
replicated Cumulus servers are typically co-deployed on the same nodes as the BlobSeer *data providers*. We used a BlobSeer chunk size of 32 MB, as previous evaluations of BlobSeer have shown this value enables the system to sustain a high-throughput for multiple concurrent data transfers.

10.3.2 Upload/download performance

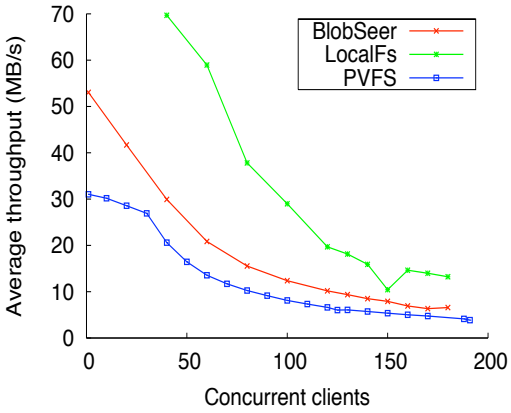
We compared the BlobSeer-based storage backend with the local file system, by measuring the throughput of data transfer operations into Cumulus when the size of the transfered file varies.

In each experiment, we used one node for the Cumulus server. The deployment configuration for the BlobSeer system included 1 *data provider* and 1 *metadata provider*. We employed standard tools developed for Amazon S3 to upload and download data to/from Cumulus, namely the s3cmd [92] command line S3 client. The throughput was computed by dividing the file size by the time it took the client command to complete.

Figure 10.4 displays a comparison between the default backend, namely the local file system, and the BlobSeer storage backend when performing uploads and downloads for increasing file sizes. We measured the throughput for files ranging from 100 MB to 2 GB in size, as such sizes are representative for virtual machine images or datasets associated with large-scale distributed applications. In the case of the Cumulus upload operation, the BlobSeer backend outperforms the local file system of the Cumulus server by around 50%. This behavior is mainly due to the BlobSeer efficient implementation of the communication layer, which allows for a high-throughput transfer towards the *data provider* node and the *data provider*'s configuration to cache the incoming data in memory and write on the disk in a background thread. The results show an important difference between the throughput of the upload and download operations for the Cumulus default storage backend. This can be explained by the fact the two operations are implemented in different manners in Cumulus. Whereas the download operation streams the file from the storage backend to the requesting client, the implementation of the upload implies streaming the file from the client



(a) Upload average throughput for 30 Cumulus servers.



(b) Download average throughput for 30 Cumulus servers.

Figure 10.5: Cumulus storage backend comparison under concurrent accesses.

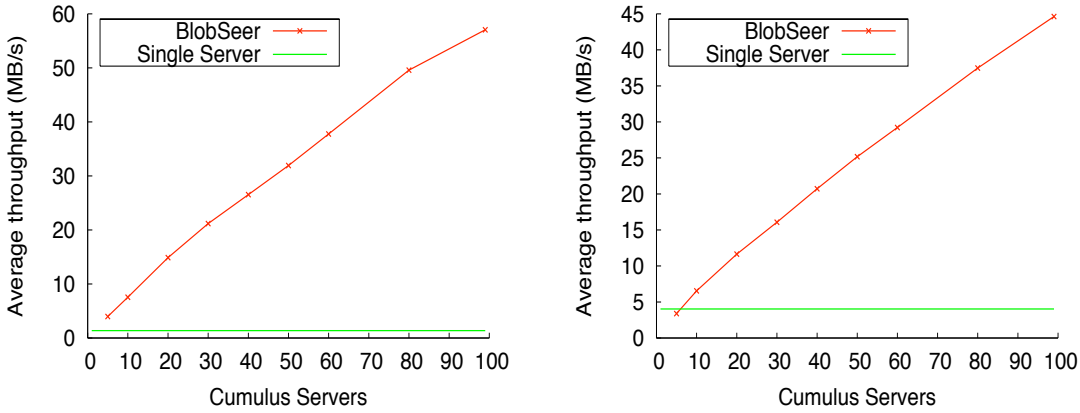
to the server and then writing it to the persistent storage.

The local file system backend is however inefficient in a distributed deployment context, as multiple Cumulus servers need to connect to the same storage system in order to provide clients with a consistent image of the stored data. The comparison on Figure 10.4 serves two purposes. First, it shows that BlobSeer is able to sustain a constant throughput regardless of the file size. Additionally, the BlobSeer backend is more efficient than using the local disk in the case of the upload operation. In the case of downloads, as in BlobSeer data is striped into 32 MB chunks, the client library has to retrieve the metadata associated with a specific file, and only after that it proceeds with the download. This behavior accounts for the download overhead observed when comparing to the default backend.

10.3.3 Scalability evaluation

To asses the impact of the BlobSeer-based backend on the scalability of the Cumulus service, we performed a set of experiments involving replicated Cumulus servers backed by the same BlobSeer instance and a large number of concurrent clients accessing the service.

Storage backend comparison under concurrent accesses. In the first experiment, we aim at evaluating the performance of a multiple-server Cumulus deployment when increasing the number of clients simultaneously transferring data. To this end, we perform a comparison between several storage backends. First, we use the local disk of each Cumulus server as the storage backend. While this approach does not store data in a distributed fashion and therefore cannot be used for a real-life application, we included this evaluation as a baseline against which to assess the performance of the other backends. The second storage system employed is PVFS [49], the parallel file system introduced in Chapter 3. PVFS allows multiple nodes to mount the same file system and to efficiently stripe and store data in a distributed manner. Finally, we evaluate the BlobSeer-based Cumulus storage in a distributed deployment configuration.



(a) Upload average throughput. (b) Download average throughput.

Figure 10.6: Data transfer rate for 100 concurrent clients when increasing the number of Cumulus Servers.

Both PVFS and BlobSeer were deployed in a similar configuration, involving 30 data storage nodes and 10 metadata nodes. The replicated Cumulus servers were deployed on the data storage nodes. In the case of PVFS, the data storage nodes were also used to mount the file system clients, while each Cumulus server used its default backend implementation to access them.

For each execution we measured the average throughput achieved when multiple concurrent clients perform the same operation on the Cumulus service. The clients are launched simultaneously on dedicated machines. Each client performs an upload and a download for a single file of 1 GB. We increased the number of concurrent clients from 1 to 100 and we measured the average throughput of each operation. The results are shown on Figure 10.5. As expected, when the Cumulus servers use their local disk for storage, the performance of the data transfers is better than when Cumulus is backed by a distributed file system. As the number of clients increases, the available network bandwidth is divided among the concurrent requests, resulting in lower average throughputs. The discrepancy between the results for the local disk and those for the distributed file systems is significant for a small number of clients. Nevertheless, when more than 100 clients transfer data simultaneously, this gap is reduced as the overhead of performing concurrent writes to the same disk becomes more important.

Despite the fact that the number of concurrent clients reaches 6 times the number of data storage nodes (which amounts to 30 nodes) in BlobSeer, the system sustains an almost constant transfer rate for more than 60 clients. This is a consequence of the efficient design that enables BlobSeer to scale to a large number of concurrent clients, regardless of the large data size involved: in this experiment, the transfer data reaches 180 GB. Moreover, BlobSeer outperforms PVFS, maintaining a throughput approximatively 30% higher in the case of uploads and 60% higher for downloads.

Impact of the server replication factor on the performance. The goal of this experiment is to assess the data-transfer performance from another point of view. We compare the perfor-

mance achieved for the BlobSeer storage backend when the number of deployed Cumulus servers varies. In this scenario, we used a fixed configuration for BlobSeer, namely 100 *data providers* and 15 *metadata providers*. We fixed the number of concurrent Cumulus clients as well, maintaining 100 simultaneous data transfers of 1 GB files for each test. We increased the number of Cumulus servers from 1 to 100 and we measured the average throughput of the client operations, for both upload and download. Figure 10.6 displays the results obtained for the BlobSeer backend and for the case of a single Cumulus server backed by the local file system. The single server values provide a baseline for assessing the speedup introduced by replicating the servers and using a distributed file system as a backend. The measured bandwidth of the 100 concurrent clients increases steadily as more Cumulus servers are introduced, suggesting that the BlobSeer-based backend scales well with respect to the number of servers connected to it.

10.4 Summary

The Cumulus service is designed to provide efficient data management for the scientific community. Our goal was to enable Cumulus to harness BlobSeer's efficient data storage and retrieval mechanisms. More specifically, we implemented a BlobSeer-based storage backend for Cumulus, so as to build a Cloud storage service that features two important properties. First, it encompasses a state-of-the-art Cloud client interface, namely the S3 REST interface of Cumulus. Second, we aimed at providing support for efficient distributed data storage and high-throughput data transfers under concurrency by leveraging BlobSeer's capabilities. We performed a set of experimental evaluations that assess the scalability of multiple Cumulus servers backed by a distributed storage solution. Furthermore, we showed the BlobSeer-based Cumulus backend is a valuable candidate for providing high-throughput, reliable Cloud data services in the context of highly-concurrent data access patterns.

Additionally, the Cumulus service also faces a set of challenges that can be overcome by leveraging the BlobSeer system. One of them is introducing *versioning support*, which can be easily enforced at the level of the storage backend, as versioning is one of the key design choices of BlobSeer.

Part IV

Evaluation with large-scale applications in Clouds

Chapter 11

MapReduce applications: impact on cost and performance

Contents

MapReduce has recently emerged as a powerful paradigm that enables rapid implementation of a wide range of distributed data-intensive applications. In this chapter, we investigate several aspects related to the execution of MapReduce applications in Cloud environments: the overhead penalty of executing MapReduce jobs in the Cloud, compared to executing them in a Grid, a cost-evaluation of the rented Cloud resources, and finally, the impact of the storage solutions employed for the input and output data. Our first goal is achieved by comparing the runtime of two MapReduce applications in specific execution environments: first on clusters belonging to the Grid'5000 platform, then in a Nimbus Cloud. Next, we consider the payment scheme used by Amazon for the rental of their Cloud resources and we estimate the cost of using our Nimbus deployment for running MapReduce applications.

11.1 The MapReduce paradigm

MapReduce [23] was introduced by Google as a solution to the need to process datasets up to multiple terabytes in size on a daily basis. An open-source implementation of the MapReduce model proposed by Google is provided within the *Hadoop* project [113], whose popularity rapidly increased over the past years.

The MapReduce paradigm has recently been adopted by the Cloud computing community as a solution for data-intensive, Cloud-based applications. Cloud providers introduced specific services to provide support for MapReduce computations, which take advantage of the huge processing and storage capabilities the Cloud holds, but at the same time, to provide the user with a clean and easy-to-use interface. There are several options for running MapReduce applications in Clouds: renting Cloud resources and deploying a cluster of virtualized Hadoop instances on top of them, using the MapReduce service some Clouds provide, or using MapReduce frameworks built on Cloud services. Amazon released *Elastic MapReduce* [118], a web service based on Hadoop, which enables users to easily and cost-effectively process large amounts of data. *AzureMapReduce* [46] is an implementation of the MapReduce programming model, based on the infrastructure services the Azure Cloud [90] offers. The framework uses the Azure Blob service as storage layer, a service that provides scalability, high throughput and data availability.

The MapReduce programming model is based on two functions specified by the user: *map* and *reduce*, which are executed in parallel on multiple machines. The *map* part parses key/value pairs and passes them as input to the *reduce* function. Issues such as data splitting, task scheduling and fault tolerance are dealt with by the MapReduce framework in a user-transparent manner.

In general, the MapReduce model is appropriate for a large class of data-intensive applications which need to perform computations on large amounts of data. This type of applications can be expressed as computations that are executed in parallel on a large number of nodes. Many data-intensive applications belonging to various domains (from Internet services and data mining to bioinformatics, astronomy etc.) can be modeled using the MapReduce paradigm. In this chapter, we consider two representative types of MapReduce applications as case studies.

Distributed grep. The *grep* application takes a huge text given as input file. It searches for a specific pattern and outputs the occurrences of that pattern. This application is a distributed job that is data-intensive only in the *map* part of the job, as the *map* function is the phase that does the actual processing of the input file and pattern matching. In contrast, the *reduce* phase simply aggregates the data produced by the *map* step, thus processing and generating far less data than the first phase of the computation.

Distributed sort. The goal of this application is to sort key/value pairs from the input data. The *map* function parses key/value pairs according to the job's specified input format, and emits them as intermediate pairs. On the *reduce* side, the computation is trivial, as the intermediate key/value pairs are output as the final result. The *sort* application generates the same amount of data as the provided input, which accounts for an output of significant size. *Distributed sort* is both read-intensive, as the mappers parse key/value pairs and sort them by key, and write-intensive, in the *reduce* phase

when the output is written to the distributed file system. Because of its both read- and write-intensive nature, this application is used as the standard test for benchmarking MapReduce frameworks.

11.2 Motivation

Our goal is to analyze the cost of moving MapReduce applications to the Cloud, in order to assess the proper trade-off between cost and performance for this class of applications. More specifically, we address the following aspects:

Virtualization overhead. We aim at evaluating the potential benefits of porting MapReduce applications onto a Cloud, with respect to the performance overhead incurred when replacing the typical MapReduce execution environment, i.e., a physical cluster, with a virtualized one, i.e., Cloud resources. The main advantage of using *virtualization* is that one can create a homogeneous environment comprising a substantial number of machines by using a considerably smaller number of physical machines. Despite the significant processing and storage capabilities offered by Cloud platforms, it is also essential to understand the requirements and features of MapReduce applications when selecting the execution environment.

Cost evaluation. Each MapReduce application entails specific requirements in terms of computation, storage and access to data. It is therefore important to study the behavior of various applications, as it may represent a crucial factor influencing the execution costs in the context of the *pay-per-use* Cloud model. To this end, we provide a performance estimation of running such applications in the Cloud and we analyze the factors that may substantially impact the cost. In particular, we focus on storage and data transfer costs, as they can prevail over the computational costs for data-intensive applications.

11.3 Computational and Cost Model

Amazon's EC2 is one of the most widely used Infrastructure-as-a-Service (IaaS) Cloud platform and the S3 [91] interface has become the "de facto" standard for transferring data at the IaaS level. Thus, in order to estimate the cost of moving MapReduce applications in the Cloud, we take the Amazon services as the reference model. Amazon users rent compute or storage resources, which are charged on a pay-per-use basis. An Amazon EC2 usage scenario consists in first selecting a virtual machine (VM) image type and then, in deploying multiple instances of this image. Amazon's cost model involves 3 types of costs.

Computational cost. The CPU cost depends on the VM type, the number of required instances and the duration of their use in EC2.

Data storage costs. They involve charges for persistently storing input and output data for the executed applications. We only focus on saving data directly into S3 objects, since existing storage alternatives, such as EBS [7] volumes, eventually rely on S3 for backup storage and introduce additional costs.

Data transfer charges include costs for moving data into and out of the Cloud. Data transfers between instances are free of charge, as well as transfers between S3 and the rented EC2 VMs. However, the transfer time for large data introduces another type of data-transfer cost, namely the computational cost to keep alive all the VMs until data transfers are completed.

To evaluate the computational cost of our experiments, we consider the *c1.medium* Amazon image type, as it meets two requirements: first, it is equivalent to the physical nodes we used when measuring the overhead of moving applications to the Cloud. Second, in [55], the authors show that the *c1.medium* image is the most cost-effective Amazon instance. The *c1.medium* instance is charged $0.19 per hour in the EU Amazon region and it features 1.7 GB of memory, 2 virtual cores and 350 GB of instance storage.

One important parameter that influences the cost-analysis of a Cloud application is the granularity at which the Cloud provider charges for resources. In the case of Amazon EC2, the rented instances are charged by the hour. This assumption may conceal the differences between storage backends or the benefits of adding resources to improve the runtime performance, when the execution lasts less than one hour. To characterize the costs associated with our experiments more accurately, we will assume per-second charges in our cost model, by dividing the hourly prices in Amazon EC2 by 3600.

Regarding the storage costs, we consider Amazon S3 charges for the EU region, i.e., $0.140 per GB for the first TB per month. As for the transfer costs, Amazon charges only for data transfers out of the Cloud, that is $0.12 per GB for data downloaded from the Cloud (download is free for less than 1 GB of output data). Some applications may need to persistently store all the input and output data in S3, as input data sets may be processed several times by the application, and output results may be further refined. Besides storage costs, Amazon S3 also charges for HTTP requests, as follows: the price for PUT, COPY, POST, or LIST requests is $0.01 per 1,000 requests and GET requests are charged $0.01 per 10,000 requests.

11.4 Evaluation

11.4.1 Execution environment

To analyze the performance and costs of MapReduce applications, we performed experiments on two different platforms. To evaluate the typical performance of MapReduce access patterns, we relied on the Grid'5000 platform, which provides the representative execution environment for MapReduce. Next, we carried out the same evaluations in an IaaS Cloud, namely the Nimbus Cloud, deployed on top of the Grid'5000 testbed. As a MapReduce framework for running our computations, we chose Hadoop, the widely used open-source implementation of Google's MapReduce model.

11.4.1.1 HDFS

The Hadoop Distributed File System (HDFS) [95] is the default storage layer of the Hadoop framework. It was designed for MapReduce workloads, providing storage for *huge files*, *fine-grained access* to data, and *high throughput* for data transfers.

In HDFS, data are organized into files and directories, each file being split into equally-sized *chunks* (typically 64 MB in size). The architecture of HDFS follows the master-slave model, featuring a centralized *namenode* in charge of managing file metadata, and a set of *datanodes* that store file chunks. To achieve the high performance and reliability required for large MapReduce jobs, HDFS also implements chunk-level replication, data-location aware scheduling and direct I/O operations between clients and *datanodes*.

11.4.1.2 Hadoop

The Hadoop [113] project includes a large variety of tools for distributed applications. The core of the Hadoop project consists of HDFS and *Hadoop MapReduce*, an open-source implementation of the MapReduce programming paradigm. The architecture of the Hadoop MapReduce framework is designed following the master-slave model, comprising a centralized *jobtracker* in charge of coordinating several *tasktrackers*. Usually, these entities are deployed on the same cluster nodes as HDFS. When a user submits a MapReduce job for execution, the framework splits the input data residing in HDFS, into equally-sized chunks. Hadoop divides each MapReduce job into a set of tasks. Each chunk of input is processed by a map task, executed by the tasktrackers. After all the map tasks have successfully completed, the tasktrackers execute the reduce function on the map output data. The final result is stored into the distributed file system acting as backend storage.

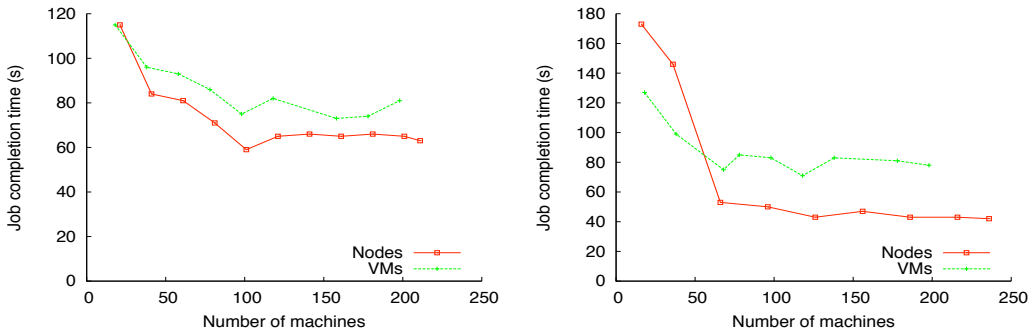
11.4.2 Virtualization overhead

The first set of experiments aims at running MapReduce applications in a typical cluster environment. For this setup, we selected the Grid'5000 cluster in Orsay, i.e., 220 nodes outfitted with dual-core x86_64 CPUs and 2 GB of RAM. Intra-cluster communication is done through a 1 Gbps Ethernet network.

The second type of environment is Cloud-oriented: it was achieved by first deploying the Nimbus Cloud toolkit on top of Grid'5000 physical nodes, and then by deploying VMs inside the resulting Nimbus Cloud. For these experiments, we used 130 nodes belonging to the Rennes site, and a VM type with features similar to the ones exhibited by the nodes from the first setup. Thus, we deployed dual-core VMs with 2 GB of RAM each.

In both setups, we create and deployed an execution environment for Hadoop featuring a dedicated node/VM for the *jobtracker* and another one for the *namenode*, while the rest of the nodes/VMs served as both *datanodes* and *tasktrackers*.

In this set of experiments, we compared Hadoop's performance in two scenarios corresponding to the environmental setups previously described: when running in a Grid environment and inside a Cloud. This performance evaluation is achieved by measuring the completion time of the *grep* and *sort* applications described in section 11.1, when Hadoop is deployed on an increasing number of nodes/virtual machines. In both scenarios, the application (*grep* or *sort*) takes as input a file of 12.5 GB stored in HDFS; we set the input size and then vary the number of nodes/virtual machines from 1 to 200. At each step, we deploy Hadoop on the respective number of machines and we measure the time it takes to run the *grep* and *sort* applications. Using this approach, we achieve a comparison of running the same workload in a similar setup, on the Grid and in the Cloud.



(a) Distributed Grep. (b) Distributed Sort.

Figure 11.1: Completion time when running Hadoop in the Grid and in the Cloud.

The time needed by Hadoop to run both applications when deployed on physical and then on virtual machines, is displayed on Figure 11.1. As expected, the job completion time for *distributed grep* is smaller than for *distributed sort*, mainly because the latter has to generate a much larger amount of output in the *reduce* phase. Also the runtime decreases when more physical /virtual machines are added to Hadoop's deployment, for both applications, and in both environments. However, the performance does not improve further when using more than a certain number of machines, around 100 in our case. This behavior can be explained by the fact that for our input data size (i.e., 12.5 GB), the optimal number for deploying Hadoop is 100 machines: each mapper processes a 64 MB chunk, therefore 200 mappers are started, 2 per machine. Thus, careful consideration must be given to fine tuning the MapReduce platform, so as to optimize the setup for a given job.

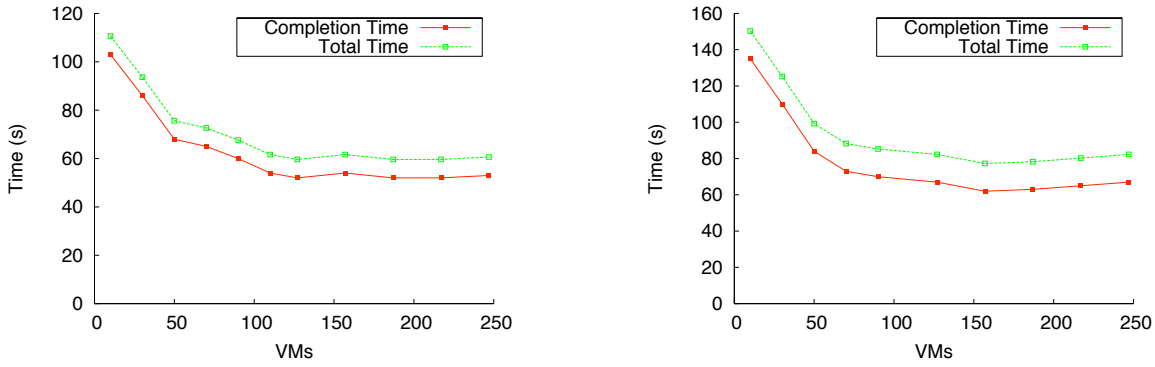
The results on Figure 11.1 also provide an assessment of the overhead of running Hadoop in the Nimbus Cloud; this overhead is however of little significance when considering the major benefit provided by the Cloud through virtualization. With a much smaller number of nodes, we managed to create inside the Cloud the same setup as the testing environment provided by a large number of physical machines in the Grid.

11.4.3 Cost analysis

In this section, we evaluate the completion time for the *grep* and *sort* applications in a Cloud environment. We also compare it against the total time to execute the application and persistently store the results in the Cloud.

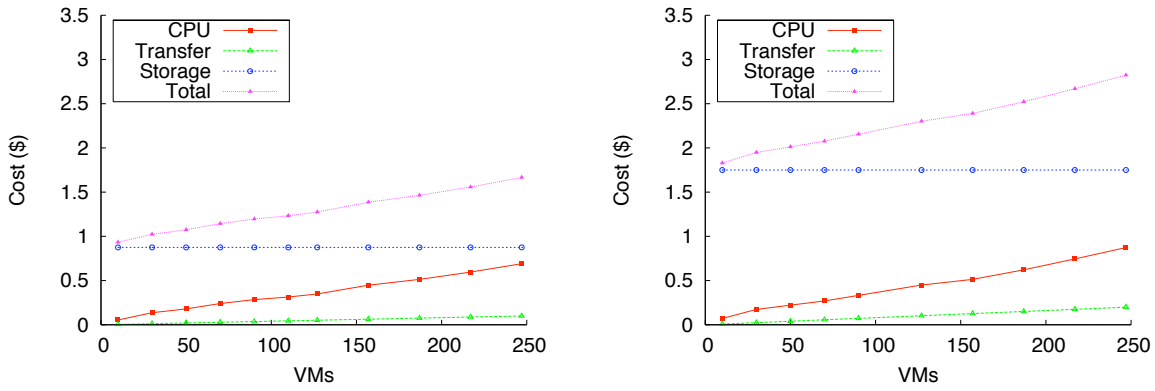
We executed two large MapReduce jobs in the same Cloud environment, each of them processing a input dataset of 100 chunks, i.e., 6.25 GB. We increased the number of VMs comprising the virtual Hadoop cluster from 10 to 250. Figure 11.2 illustrates the completion time for both *grep* and *sort* applications. As in the previous experiment, the runtime improves when more VMs are added to the virtual cluster. However, the graph flattens out as the number of VMs reaches 50 and thus the framework deploys a number of mappers equal to the number of input data chunks.

We plotted in the same figure the total time it takes the application to download the input data from an S3-based Cloud storage service, to complete its execution and to upload the



(a) Distributed Grep. (b) Distributed Sort.

Figure 11.2: Completion time and total execution time (including data transfer time) when running Hadoop in the Cloud.



(a) Distributed Grep. (b) Distributed Sort.

Figure 11.3: Cost evaluation.

results back into the Cloud. We employed a Cumulus repository as the data Cloud service, using the following configuration: 50 Cumulus servers backed by a BlobSeer-based storage layer comprising 50 *data providers* and 10 *metadata providers*. The Cumulus servers were co-deployed with the BlobSeer *data providers*, on dedicated machines. We performed a synthetic experiment to determine the Cumulus upload and download time for 100 chunks. The input of both applications amounts to 100 chunks of 64 MB each, whereas their respective output is different: *grep* yields a negligible output and *sort* generates the same amount of data as its input, i.e., 100 chunks. We consider a set of 100 parallel processes, each of them downloading 1 chunk from the S3-based service into HDFS and 100 parallel processes for performing the upload operation from HDFS to S3. We measured the upload and download times and we added them to the execution time of each application. We computed the download time as the time the slowest client needed to complete its operation, as the VMs cannot start processing input data before the full dataset is transferred. In a similar way, all nodes have to be running while output data is uploaded; hence, the upload time is the duration of the slowest chunk upload. The results in Figure 11.2 show the transfer time is less significant

than the computation time, especially because the transfers are done in parallel, the transfer time accounting just for uploading 1 chunk.

Next, we evaluated the cost of running the *grep* and *sort* applications in the Cloud. We analyzed the types of costs detailed in section 11.3, namely the cost of running VM instances, the cost of storing the data into S3 and the cost of data transfers. Since data transfers between S3 and the Cloud environment are free, they do not impact the total cost as storage expenses. However, data transfers have to be accounted for in terms of time, as the running VMs cannot be stopped until all the output data has been persistently stored on a Cloud data service.

Figure 11.3(a) shows the costs of running the *grep* application as a function of the number of virtual machines that act as Hadoop nodes, for the same input size as in the previous experiment. The computational cost is defined as the number of VMs $\times$ the execution time $\times$ the cost of the VM instance per second. It increases as more virtual machines are provisioned, since the cost of deploying additional VMs outweighs the performance gain caused by the decreasing execution time. The data transfer cost is computed as the slowest transfer time of 1 chunk $\times$ the number of VMs $\times$ the cost of one VM instance per second. As *sort* processes the same amount of input data as *grep*, but instead it generates a large output, the transfer costs in Figure 11.3 appear to impact more on the total cost for running the *sort* application.

The data storage costs do not vary with the number of deployed VMs, since the input and output data are the same at each deployment setup. Applications that process small amounts of data have negligible data transfer costs. In contrast, data-intensive applications have different cost constraints: the computation time is usually less important than the time spent in I/O operations. This is also the case for the *grep* application, and consequently, the storage cost is the dominant factor in the total cost when the number of provisioned VMs is small. The cost estimation for the *sort* application is presented in Figure 11.3(b). Since we used the same input file, that is 100 chunks of text files, the results exhibit the same behaviour as for *grep*. However, the data transfer costs have a more significant impact on the total cost compared to *grep*. This is a consequence of the large output file generated by *sort*, which has to transfer the 100 output chunks into S3 as well.

11.5 Related Work

Several studies have investigated the performance of various Cloud platforms and the costs and benefits of running scientific applications in such environments. Most evaluations focused on the Amazon's EC2 Cloud, as it has become the most popular Infrastructure-as-a-Service (IaaS) platform and has imposed its specific cost model to the Cloud computing community. The works of Walker [111], Evangelinos [27] and Hill [51] explored the tradeoffs of running high-performance applications on EC2, showing that the Cloud environments introduce a significant overhead for parallel applications compared to local clusters. The cost of using HPC Cloud resources is discussed by Carlyle and Harrell [16], who introduced a cost model for local resources and compared the computational cost of jobs against a Cloud environment. However, this work includes only a benchmark-based performance evaluation and no specific type of application is considered.

Several works [25, 55, 11] have focused on loosely-coupled applications, where the authors conducted a cost analysis of running scientific workflows in Cloud environments.

They considered the performance penalties introduced by Cloud frameworks and evaluated computational and storage costs through simulations and experiments on EC2 and a local HPC cluster. More in-depth studies have investigated data storage in Clouds, evaluating the Amazon S3 service through data-intensive benchmarks [84]. Moreover, in 2010, Juve [55] evaluated several file systems as Cloud storage backends for workflow applications, emphasizing running times and costs for each backend. Other works [59] conducted a comparative evaluation of Cloud platforms against Desktop Grids. They examined performance and cost issues for specific volunteer computing applications and discuss hybrid approaches designed to improve cost effectiveness. Gunarathne and Fox [47] introduced the AzureMapReduce platform and conducted a performance comparison of several commercial MapReduce implementations in Cloud environments. The analysis included scalability tests and cost estimations on two MapReduce applications.

11.6 Summary

In this chapter, we addressed the challenges raised by executing MapReduce applications in Cloud infrastructures instead of dedicated clusters.

We evaluated the performance delivered to the users by measuring the completion time of two MapReduce applications, *grep* and *sort*, executed on the Hadoop framework in two different settings: first, we deployed Hadoop on physical nodes in Grid'5000; then we repeated the experiments using VMs provisioned on a Nimbus Cloud. Furthermore, we evaluated the computation, data transfer and storage costs for running these applications in the Cloud, by considering Amazon services' charges as reference costs. We showed that for one application execution, the storage costs have an essential impact on the total cost. Nevertheless, persistently storing the input and output datasets in the Cloud allows various processing applications to use the same data, without incurring additional storage costs. Each application accessing an already existing dataset will only observe data transfer costs, which, however, do not represent a significant fraction of the total costs.

We employed a BlobSeer-backed Cumulus service as a data Cloud solution for persistent storage. The data transfer costs incurred by the applications directly depend on the performance of the data storage service. The low transfer costs we have obtained can therefore be correlated with the capability of a Cumulus-based service to provide high-throughput access to data. These results are consistent with the Cumulus evaluation benchmarks we presented in Chapter 10, indicating that BlobSeer supplies an efficient and cost-effective storage backend for Cloud services.

Chapter 12

Tightly-coupled HPC applications

Contents

An increasing number of studies from the parallel high-performance computing community have started to focus on investigating Cloud infrastructures as alternatives to the traditional dedicated supercomputers. In addition to the illusion of infinite computing resources promoted by Cloud providers, tightly-coupled HPC applications typically require efficient parallel storage systems and high-performance network interconnects.

This chapter discusses the means to deploy such an application in a Cloud environment and the necessary tools to interface it with existing Cloud services. Furthermore, we evaluate the impact of using Cloud storage solutions for a real-world application for atmospherical simulations, Cloud Model 1 (CM1).

12.1 Case study: the Cloud Model 1 (CM1) application

12.1.1 Application model

We target tightly-coupled, high-performance computing applications specific to the scientific community. Such applications exhibit a set of common features, discussed below.

Parallel processes. Generally, HPC applications split the initial problem into a set of subproblems. Then, these smaller subproblems are spread across a fixed set of processes, which handle the data in parallel. Such applications typically rely on message-parsing systems (e.g., MPI) for inter-process communication and synchronization.

Compute-intensive simulations. We consider applications that simulate complex phenomena in various contexts, including high-energy physics, atmospheric simulations, earthquake simulations or satellite image processing. They usually require significant computing resources and spend more time for computing the results than for performing I/O operations.

Massive output data. Real-life simulations involve large-sized output, as they compute a set of variables describing the evolution in time of the modeled phenomenon. They are typically designed to store results and additional application logs in a parallel file system, such as GPFS [94] or PVFS [49].

No concurrent access to files. Each process computes a subset of the problem output data. Concurrently dumping results into a single shared output file may lead to I/O bottlenecks prone to decrease the overall performance of the application. Therefore, we consider applications involving independent processes, which perform write operations in separate files.

12.1.2 Zoom on CM1

Cloud Model 1 (CM1) [19] is a three-dimensional, time-dependent numerical model designed for atmospheric research, in particular for modeling major phenomena such as thunderstorms. CM1 simulates a three-dimensional spatial domain defined by a grid of coordinates specified in a configuration file. For each spatial point, the application is designed to compute a set of problem-specific variables, including wind speed, humidity, pressure or temperature.

A CM1 simulation involves computing the evolution in time of the parameter set associated with each grid point. To this end, the 3D domain is split along a two-dimensional grid and each obtained subdomain is assigned to its own process. For each time step, all processes compute the output corresponding to their subdomain, and then they exchange border values with the processes that handle neighboring subdomains. The computation phases alternate with I/O phases, when each process dumps the parameters describing its subdomain to the backend storage system.

CM1 is implemented in Fortran 95 and the communication between processes relies on MPI [43]. The output results can be stored in various formats targeted at specific scientific

communities, such as GrADS [42], netcdf [73] or HDF5 [106]. The MPI-based implementation requires each process to write data into a separate file for each time step. We employed the GrADS format in our evaluations, a binary format widely used for scientific data sets [42].

12.2 Cloud data storage for CM1

We aim at running tightly-coupled, MPI-based applications in a Cloud environment. More specifically, our goal is to assess the impact of relying on Cloud services to address the storage needs of the application. To this end, we focused on S3-compatible storage services, as Amazon's S3 protocol is the standard data-transfer solution in IaaS Clouds. The reasons behind this choice are summarized in the following section. Next, we detail our proposed solution to enable applications to access S3-based services without requiring modifications in the code.

12.2.1 Motivation

IaaS Clouds typically provide the user with a set of virtual machine instances which can host applications. Such VMs are equipped with local disks the applications can use to store generated or input data. This storage solution, however, is not persistent, as the disk is wiped out each time a virtual machine lease ends. Amazon provides services such as the Elastic Block Store (EBS) [7], which has been introduced in Section 3.3. Essentially, EBS allows users to attach a virtual disk to each of their VMs, which can then be backed up onto S3 to persistently store saved data. This solution has however a major drawback: each EBS disk corresponds to a specific virtual machine, and therefore the various VMs cannot share the stored data. Furthermore, as computing VMs carry out simulations and generate pieces of the output data on local EBS disks, no application can access the whole final results without scanning each EBS disk and possibly copying the data onto a single disk to enable further processing. To access and further process the large amounts of data generated by simulations, this approach is both inefficient and expensive in terms of resource usage and costs.

Another potential solution is to deploy a parallel file system backed by the virtual machines local disks and to use it as a storage service for the application. While this approach has the advantage of providing the application with a standard file system interface, it does not address the persistency requirement. The computed results are either lost at the end of the VM lease, or the user has to manually save them into a persistent repository, such as Amazon S3, to make them available to higher-level applications. This operation increases the time it takes for the simulation to complete, as it adds the output transferring step.

Moreover, such applications typically generate several output datasets, one for each intermediate time step of the simulation. These results serve as an input for higher-level tools. For instance, data-mining and visualization tools, such as VisIt [110], may perform real-time data analysis, debugging or data aggregation for visualizing the output at each timestep.

To address the aforementioned storage challenges, we proposed an interface module to stream application data to S3-based services, which meets the following requirements.

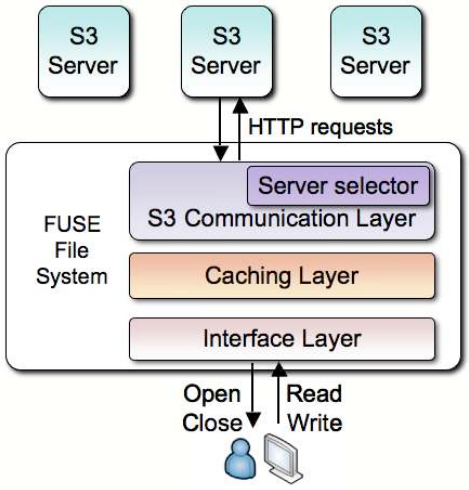


Figure 12.1: The architecture of the S3-backed file system.

File-system interface. We designed a file-system interface relying on FUSE (Filesystem in Userspace) [29]. FUSE-based file systems present the advantage of a POSIX-compatible interface, exposing data as regular files. Additionally, a traditional interface facilitates the execution of HPC applications in a Cloud environment, by avoiding any modification at the level of the I/O operations.

Backup files to S3. Each WRITE operation initiated by the application is translated into an upload to the S3 service. As a result, each simulation output file is forwarded to the persistent Cloud storage and in the same time it is made available to higher-level tools to process it as the simulation continues.

File prefetching. To optimize READ operations, we introduced a prefetching mechanism which downloads the files from the S3 repository and stores them locally to improve READ access time.

12.2.2 Designing an S3-backed file system

We designed a FUSE-based file system to enable applications that need access to standard files to interact with S3-compliant services. Our approach implements the architecture on Figure 12.1.

POSIX interface layer. This layer implements the FUSE API to provide applications with a hierarchical file-system namespace. Typically, the applications we target do not perform namespace management operations, such as directory listing or moving. Instead, they employ a flat directory structure to access input data or to store output files. Moreover, each node generates independent results, so that there is no need for file sharing between processing nodes. As a consequence, we only maintain a local directory structure for each node. However, when uploading files to the S3-based service, their full path is saved so that the namespace created by the application can be regenerated upon download.

Caching layer. In order to efficiently address the I/O needs of scientific applications, we implemented a caching layer based on memory-mapped files. The main advantage of this approach is that it improves I/O performance for both the application and the streaming mechanism that uploads or downloads the file from S3. Moreover, memory mapped files avoid expensive memory-copying operations and load only the accessed regions of the files that do not fit into memory. The *Interface Layer* forwards the file operations to the *Caching layer*, as it implements the file-management mechanism.

S3 Communication layer. It is responsible for file uploads and downloads to/from an S3-compatible service. Each time a file is closed, the *Caching Layer* verifies whether it has been locally modified. If changes are detected, the file is streamed to the S3 service. Conversely, when an application tries to access a file that is not present on the local file system, the file is prefetched from the S3 service and made available to the application in the form of a local file.

Server selector. This module provides the *Communication Layer* with an S3 server to use for data transfers. It requires a list of available S3 servers and it employs a hash function to select one of them. This approach favors a uniform distribution of the VMs among the existing S3 servers.

The S3-backed file system was implemented in Python, as it provides efficient FUSE libraries for the file-system interface. Moreover, we employed the *boto* libraries [104], a set of widely popular interfaces for accessing Amazon-compatible services.

12.3 Evaluation

To assess the performance of the various Cloud backends employed for CM1, we conducted a set of experiments on Grid'5000. We relied on a Nimbus Cloud to provide the IaaS environment on which we deployed virtual clusters. Each virtual machine of such a cluster is equipped with the S3-backed file system. The CM1 application is configured to execute in parallel on the virtual cluster nodes, and to store output data into the FUSE-based file system. The file system is backed by a large deployment of Cumulus servers, which rely on several backends to store data in a distributed fashion.

The experiments we performed focus on analyzing the performance and scalability of a tightly-coupled application such as CM1 in a Cloud environment, in two different contexts. First, we investigate the behavior of the application when the size of the deployment increases, by maintaining the size of the processed domain constant for each process. Second, we fix the size of the initial domain and we increase the number of processes, so as to observe the resulting speedup as the domain is split in more subdomains.

12.3.1 Experimental setup

We used 50 nodes to deploy replicated Cumulus servers on top of three storage backends: the local file system on each Cumulus node, BlobSeer and PVFS. Both BlobSeer and PVFS employ 50 *data providers* and 10 *metadata nodes*, each of them being deployed on dedicated machines. Another 64 nodes were used to deploy a Nimbus cloud that enabled us to execute the CM1 application in large virtual clusters. For each experiment we created Nimbus

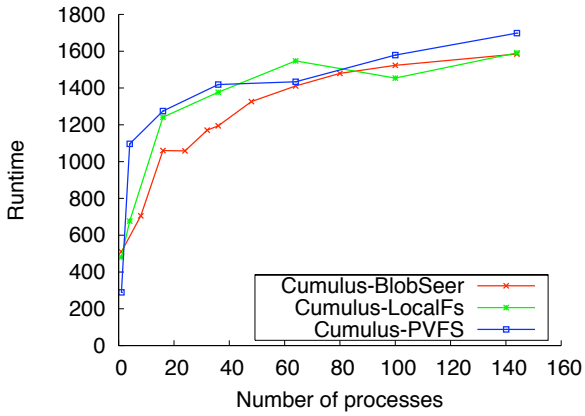


Figure 12.2: Application runtime for 10 minutes of simulated time with a 20 s time step.

virtual clusters of quadcore virtual machines with 4 GB of RAM. Each virtual machine is equipped with the S3-backed file-system module, to enable CM1 to directly store its output data in Cumulus. Each file-system module is provided with the list of running Cumulus servers. Each virtual machine interacts with a single Cumulus server, selected through a hash function to favor a uniform distribution of the connections.

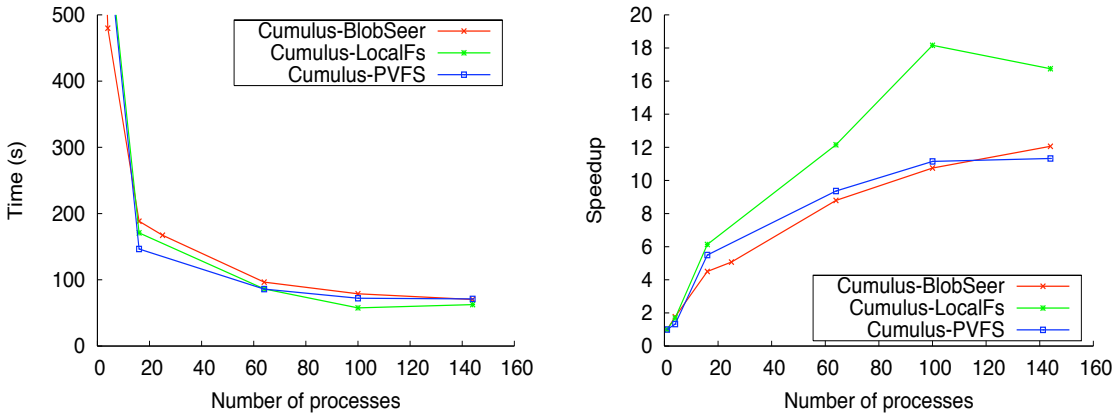
The Cloud Model1 (CM1) application is representative for a wide class of applications that simulate the evolution of a phenomenon in time. Each simulation is associated with a 3D domain and a time interval. A simulation consists in obtaining the values for a set of parameters for each point of the domain and for each time step. The initial domain is split among the processes and each of them is in charge of a particular subdomain.

We used a 3D hurricane simulation described in [15]. The application was configured to use MPI to split the initial domain and perform the simulation in a distributed fashion. It generates a set of output files for each MPI process and for each time step. We chose to compute all the simulated parameters, so as to obtain the largest possible output.

12.3.2 Completion time when increasing the pressure on the storage system

For the first experiment, we executed the application for 10 minutes of simulated time, with a time step of 20 seconds. The initial domain consists of 200 $\times$ 200 points describing a grid of squares with an edge length of 15,000 m. We increase the number of MPI processes but we maintain the same number of points associated with each process by increasing the precision of the simulation (i.e. decreasing the size of the squares in the initial domain). We generate output files each 2.5 minutes. As the total simulated time is 10 minutes, we obtain 4 output files, each of them of 85 MB in size, amounting to 340 MB generated by each process per run.

We deployed 4 MPI processes on each virtual machine (one for each core) and increased the number of processes from 1 to 144. The total size of the data generated for these simulation increases from 340 MB to 50 GB. Figure 12.2 shows the simulation completion time when increasing the number of processes when output data is stored into Cumulus. The purpose of this experiment is to assess the overhead introduced by the data storage service when the number of concurrent clients increases. To this end, we maintained a constant size of the simulated domain and the size of the output data per process for each point of the



(a) Completion time.

(b) Speedup.

Figure 12.3: Storage backend comparison for 30 minutes of simulated time with a 20 s time step.

graph.

The results show that completion time increases with the number of processes, the graph featuring a very steep increase when the number of processes is small. However, for more than 20 processes, the curve flattens for all three storage solutions. This behavior suggests the application is able to scale despite storing output data into an external repository such as Cumulus.

On the one hand, the runtime increasing trend can be explained by the larger resolution of the simulation, which leads to a larger time spent in communication among MPI processes, as there are more border values to exchange. On the other hand, the increasing pressure on the storage backend of Cumulus introduces a performance penalty as well, contributing to the increase of the execution time.

The results indicate the BlobSeer and the PVFS backends for the Cumulus servers do not lead to a performance drop for the application that stores data into Cumulus, when comparing against the local file system backend. Furthermore, the BlobSeer and the PVFS backend layers provide distributed storage solutions.

Moreover, the BlobSeer-based Cumulus version slightly outperforms the PVFS Cumulus backend. Similarly to the Cumulus benchmarks in Chapter 10, this result confirms the higher throughput delivered by BlobSeer in this context.

12.3.3 Application speedup

This experiment aims at evaluating the speedup obtained by scaling up the number of application processes for the same initial problem. We provide a comparison of various storage backends prone to impact on the application performance and consequently, on the improvement achieved when increasing the number of processing nodes for the same problem size.

For this evaluation, CM1 was executed for 30 minutes of simulation time, with a timestep of 20 seconds between consecutive computations. In contrast to the previous experiment, we increase the duration of the computation phase, so as to highlight the significance of

adding more processing nodes. Hence, each simulation consists of 90 processing steps. The frequency of output generation is of 450 seconds, which is equivalent with 5 output files per process and per simulation for the 30 minutes of total simulation time.

The size of the simulated spatial domain is 200 $\times$ 200 points, each of them delimiting 15,000 m of simulated surface. Figure 12.3(a) displays the completion time of the application when increasing the number of processes. For each point on the graph, we maintain the total size and number of points of the simulated 3D domain and we divide the domain by the number of available processes. Therefore, Figure 12.3(a) depicts the time it takes the application to complete when the initial problem is divided among an increasing number of processes. Additionally, we have shown the corresponding speedup in Figure 12.3(b). The speedup for a specific number of processes is computed as the measured execution time of the application for a single process divided by the execution time when all the processes are employed.

As expected, as we divide the simulated spatial domain among an increasing number of processes, the application completes its execution much faster. The drop in the execution time is a consequence of the smaller number of points each process is in charge of simulating, and thus of the diminished number of border values to be exchanged between processes and output data to be sent to storage backend. The obtained performance of the three backends is similar, as most of the execution time accounts for computation. The computation time is particularly important in this experiment, as we carry out 90 simulation timesteps. In contrast, in the previous experiment we performed only 30 simulation steps and we focused on generating large amounts of output data to assess the impact of the employed storage system. The speedup measured for the two distributed backends is similar. We however obtained a better speedup when using the local file system of each Cumulus server as a storage soution. This result is mainly due to the large execution time detected for the local file system in the case of only one process, when the all generated data had to be dumped on a single node's file system, whereas it was distributed among storage nodes for the two other backends.

12.4 Summary

In this chapter we addressed the problem of employing virtual clusters provisioned from IaaS Clouds for scientific, tightly-coupled applications. We discussed one of the most important issues that may limit the adoption of the Cloud paradigm for this type of applications, namely data management.

We relied on an atmospheric phenomena modeling application to conduct a set of evaluations in a Nimbus Cloud environment. This application is representative for a large class of simulators that compute the evolution in time set of parameters corresponding to specific points in a spatial domain. As a consequence, such applications generate important amounts of output data. We evaluated an S3-compliant Cloud storage service as a storage solution for the generated data. To this end, we employed distributed Cumulus services backed by various storage systems. The reason for targeting this approach is that storing output data directly into the Cloud as the application progresses can benefit higher-level applications that further process such simulation data. As an example, visualization tools need to have real-time access to output data for analysis and filtering purposes.

We built an interfacing module to enable the application to run unmodified in a Cloud environment and to send output data to an S3-based Cloud service. Our experiments show that distributed Cumulus backends, such as BlobSeer or PVFS, sustain a constant throughput even when the number of application processes that concurrently generate data becomes 3 times higher than the number of storage nodes.

Part V

Conclusions and perspectives

Chapter 13

Conclusions

Contents

Important academic and industrial actors, such as Google, Amazon or Yahoo!, have recently started to investigate Cloud computing, an emerging paradigm for managing computing resources. The Cloud computing model targets a broad range of applications, focusing on providing cost-effective support for large-scale distributed applications, which typically require expensive dedicated data centers. As data volumes generated and processed by such applications increase, a key requirement that directly impacts the adoption rate of the Cloud paradigm is efficient and reliable data management.

In this manuscript we addressed the problem of building an autonomic, efficient and secure storage service for Cloud environments by leveraging BlobSeer, a large-scale distributed data-management platform.

The contributions of this manuscript can be summarized as follows:

Self-management aspects in storage systems. We proposed a set of self-management mechanisms targeted towards data- management systems. We analyzed the requirements of large-scale storage systems with respect to autonomic properties and we proposed global architectures to equip such systems with several self-* capabilities. The next step was to validate each self-management solution for a specific storage system. To this end, we selected BlobSeer, a distributed storage system designed to handle massive data and to provide high-throughput operations under heavy concurrency. The achieved results are detailed below:

Introspection mechanisms enabling self-awareness. The first step towards an autonomic data-sharing system is to equip the system with introspection capabilities. We introduced a layered architecture enabling self-awareness at the level of data-storage systems through monitoring and processing system-specific parameters. Data-gathering tasks can be carried out by any monitoring system that exhibits a set of prerequisite

properties, such as scalability and flexibility with respect to the type of collected data. Nevertheless, the monitored parameters, as well as the processed data yielded by the introspection architecture depend on the needs and the applicative context of the monitored storage system. To validate our architecture, we enhanced the BlobSeer system with self-awareness, by defining relevant monitoring parameters and adapting the introspection architecture accordingly. Furthermore, we built a visualization tool that exploits the output of the self-awareness component to provide aggregated or detailed views describing the state of the BlobSeer system and the distribution and access patterns of the stored data.

Generic security framework for self-protection. We proposed a generic security framework allowing administrators of data- management systems to define and automatically enforce complex security policies. This security framework is designed to be independent of the target system. Thus, it only requires access to specific monitoring data describing the system user's actions. The security framework includes a policy definition component that allows administrators to define security policies for a broad range of attacks. The employed format is flexible and extensible, supporting complex policies that can take into account various user actions linked through temporal dependencies. The proposed framework analyzes each policy and triggers a feedback action whenever the attack scenario described by the policy matches the real user activity monitored from the system. The feedback modules can be adapted for the target data-storage system, as they range from blocking the detected malicious user to modifying the types of services it has access to. We equipped BlobSeer with self-protection properties by interfacing it with the security framework and we evaluated its reactivity and intrusiveness by means of Grid'5000 experiments featuring tens of concurrent malicious clients.

Self-configuration through dynamic dimensioning. In the context of Cloud data management, self-configuration plays a significant role in providing efficient services, while maintaining a cost-effective deployment scheme. We developed a self-configuration architecture aiming at automatically expanding or contracting the pool of running storage servers, so as to optimize the deployment scale of a distributed storage system with respect to the real-time workload. This self-configuration mechanism requires access to two system-specific services: first, it needs to have access to information describing data accesses and load at the level of the storage nodes; second, it has to be interfaced with a replication tool that continuously checks and restores the replication degree of the stored data. We built a functional implementation of this architecture for the Blobseer system, enabling it to automatically scale in or out the number of storage providers, according to specific rules defined by the administrator.

Designing a BlobSeer-based file system for Cloud storage. The second part of our contribution was dedicated to designing a Cloud storage service able to provide Cloud standard interfaces, while exploiting all the advantages brought by BlobSeer in the area of large-scale distributed storage. To achieve this goal, we developed a file system layer on top of BlobSeer, which exposes a hierarchical file namespace enhanced with the concurrency-optimized BlobSeer primitives. Furthermore, we integrated the BlobSeer file system as a backend for Cumulus, an efficient open-source Cloud storage service. We validated our approach through

extensive evaluations performed on Grid'5000. We devised a set of synthetic benchmarks to measure the performance and scalability of the Cumulus system backed by BlobSeer, showing it can sustain high-thorughput data transfers for up to 200 concurrent clients.

Evaluating Cloud storage solutions with real-life applications. To explore the advantages and drawbacks of employing Cloud storage services for distributed applications that manage massive amounts of data, we investigated two types of applications. The experiments were performed on top of a Nimbus Cloud deployed on physical resources supplied by Grid'5000.

First, we carried out a series of evaluations involving typical MapReduce applications, to assess overhead of moving MapReduce frameworks from the context for which they were designed, i.e., cluster environments, to Cloud virtual clusters. The obtained results showed the performance penalty of running the MapReduce applications in the Cloud is not significant compared to the efficient resource utilization achieved by using virtualization. Additionally, we performed a cost evaluation of moving such applications into the Cloud and hosting the generated data on Cloud services, assuming a cost model similar to the pay-per-use charges enforced by Amazon services.

The second type of application we studied is an atmospheric modeling tightly- coupled application, built as a set of parallel processes that communicate through MPI. We evaluated the performance of running such an application in a Cloud environment and employing a Cloud storage service for persistently saving the output files. We relied on Cumulus for the data storage tasks and we assessed the impact of using various storage backends on the overall performance of the application. The obtained results show that distributed Cumulus backends, such as BlobSeer, sustained a constant throughput as we increased the number of concurrent processes and the amount of generated data.

Chapter 14

Perspectives

Contents

In this manuscript we addressed several challenges emerged in the area of Cloud data management. This chapter describes the new research directions that surfaced as this work was carried out.

14.1 Self-management in Clouds

In this manuscript we investigated a set of key self-* properties for Cloud data-management services. We designed generic components and we validated them by adapting them for the BlobSeer system. This work brings forward several optimization directions and opens the path for exploring new self-management aspects for Cloud services.

Complex self-protection techniques. We proposed a generic security framework to handle complex security attacks by means of configurable security policies. One key component of the security framework is responsible for managing the *trust level* of users. We plan to extend the *Trust management* module, so as to enable it to reflect user actions more accurately, by taking into account more factors when computing the *trust level* of each user. Furthermore, we intend to introduce dynamically customizable client access rights and quality of service based on trust values. This can be done by enhancing the security framework with the capability to provide adaptive security policies. Thus, the administrator could define generic security policies, which would then be automatically customized and updated for

each user, according to its trust level. In this way, specific rights can be restricted for malicious users, without affecting the rest of users that access the system and without requiring the intervention of an administrator to tune the policies.

Enhanced security in BlobSeer. To enable BlobSeer as a fully-fledged Cloud storage system, we aim to enhance it with a set of security mechanisms. As Cloud users are typically untrusted entities, our goal is to introduce adequate authentication and authorization mechanisms for BlobSeer users and to provide privacy guarantees through anonymization. To this end, we plan to focus on certificate management, encryption capabilities, as well as credential management and access control lists.

Adaptive replication. We developed a *replication manager* component to complement the self-configuration solution implemented for BlobSeer. This *replication manager* is in charge of maintaining the replication degree of data chunks upon removal of a storage server from the pool of active storage nodes. Nevertheless, this component can be enhanced with additional features, such as an adaptive modification of the replication degree of specific BLOBS, based on the load and utilization rate of the data. Such an approach would rely on the self-awareness component already integrated into BlobSeer, and would analyze collected access information to make automatic decisions regarding the optimal replication degree of each BLOB.

Self-optimization. Self-optimization addresses the improvement of a system's performance with respect to various parameters, such as resource utilization, workload management or resource allocation. In the case of BlobSeer, a self-optimizing component could primarily provide efficient allocation strategies for data chunks. Previous work in this area [71] has focused on optimizing allocation based on the load of data providers. This can be extended by integrating more in-depth knowledge about the state of the system extracted from the self-awareness component. For instance, allocation can also anticipate the usage of the new versions of specific BLOBS by analyzing the past utilization rates of the BLOB. Moreover, self-optimization components can integrate modules in charge of identifying access hot spots and efficiently redistributing data to achieve load balancing.

14.2 Optimizing Cloud data storage

We designed a BlobSeer-based file system and we integrated it as a Cumulus storage backend. This approach allowed Cumulus to exploit the efficient data management techniques provided by the BlobSeer system. Moreover, the BlobSeer backend also plays an important role as an enabling technology for several additional features described below.

New features for Cumulus. Cumulus aims at providing Amazon S3-compatible services for data storage in the Cloud. Apart from the basic data upload and download mechanisms, S3 has introduced a set of new primitives targeting large datasets: *multi-part uploads* and *versioning*. To provide similar features in an efficient manner, Cumulus needs to be backed by a storage solution designed with built-in support for such operations. By leveraging

the BlobSeer versioning-based interface, Cumulus can introduce versions without incurring any performance overhead. Furthermore, BlobSeer is specifically devised to exhibit a high-throughput under concurrent accesses to the same BLOB. Thus, building a *multi-part upload* service on top of BlobSeer would expose its efficient features at the level of the Cloud storage system.

BlobSeer-based VM storage in Nimbus. Cumulus is an important building block in the Nimbus Cloud framework, for which its fundamental role is to act as a virtual machine image repository. By employing a BlobSeer-backed Cumulus system in this context, the Nimbus framework can be enhanced with several interesting features. First, BlobSeer can be integrated into Nimbus as a VM image propagation and deployment mechanism, by leveraging the work done in [78]. This approach requires adapting the Nimbus propagation module, to enable it to support a BlobSeer-based Cumulus backend. Furthermore, the deployment mechanisms in Nimbus have to be modified to take advantage of the efficient image booting mechanism provided by the BlobSeer system. Second, storing VM images into BlobSeer can leverage another BlobSeer feature, namely Nimbus can be enhanced with the efficient multi-snapshotting capabilities described in [78].

14.3 BlobSeer-based Cloud data storage in more applicative contexts

We have enabled BlobSeer as a storage service for large datasets generated and processed by scientific applications, by integrating it with the Cumulus system. We aim at further exploring this direction and the ways to take advantage of the BlobSeer's scalable architecture and high-throughput concurrent data transfers through evaluations of various classes of applications.

Tightly-coupled applications. We plan to perform more in-depth evaluations that employ Cumulus as a storage service for tightly-cloupled scientific applications. More specifically, our goal is to conduct experiments on larger-scale deployments and to assess the impact of various types of storage, ranging from file systems deployed inside the virtual machines to Cumulus-based approaches. Furthermore, it is also important to evaluate various workloads and access patterns, as well as the advantages of leveraging Cumulus for online visualization of the generated simulation results.

Workflow applications. Several studies have tried to asses the performance of *workflow applications* in Cloud environments [25, 55]. Workflows are loosely-coupled parallel applications that comprise a set of tasks linked through data dependencies. In contrast to MPI-based applications, which communicate through message passing mechanisms, workflows typically communicate through distributed filesystems. Thus, each task writes its output files into the shared storage, from where the next tasks can access it as input. This particular access pattern is an interesting case study for Cumulus, as it requires a distributed storage solution that provides efficient read and write access to data. In this scenario, we aim at evaluating the performance of several Cumulus backends for various workflow applications.

Cost evaluations. We have conducted a set of experiments to assess the cost of two representative MapReduce applications executed in Cloud environments. We intend to widen the range of evaluated MapReduce applications and to determine the most cost-effective storage solution for each type of workload. Furthermore, we plan to extend our cost evaluations at the level of other types of distributed applications, including workflows or MPI-based applications.

Bibliography

[1] Cloud Security Alliance. http://cloudsecurityalliance.org/research/initiatives/top-threats/.

[2] Globus Grid Security Infrastructure. http://www.globus.org/security/overview.html.

[3] Grid'5000. https://www.grid5000.fr/.

[4] The Kadeploy project. http://kadeploy.imag.fr/.

[5] The OAR project. http://oar.imag.fr/.

[6] Amazon Auto Scaling. http://aws.amazon.com/autoscaling/.

[7] Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/.

[8] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[9] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46:43–48, February 2003.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37:164–177, October 2003.

[11] G.B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *2010 Sixth IEEE International Conference on e-Science Workshops*, pages 1 –7, dec. 2010.

[12] Windows Azure Blob. http://msdn.microsoft.com/en-us/library/dd179376.aspx.

[13] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[14] John Bresnahan, Kate Keahey, David LaBissoniere, and Tim Freeman. Cumulus: an open source storage cloud for science. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 25–32, New York, NY, USA, 2011. ACM.

[15] George H. Bryan and Richard Rotunno. Evaluation of an analytical model for the maximum intensity of tropical cyclones. *Journal of the Atmospheric Sciences*, 66(10):3042–3060, 2009.

[16] A.G. Carlyle, S.L. Harrell, and P.M. Smith. Cost-effective hpc: The community or the cloud? In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 169 –176, 30 2010-dec. 3 2010.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[18] B. Claudel, G. Huard, and O. Richard. Taktuk, adaptive deployment of remote executions. In *HPDC '09*, pages 91–100, New York, NY, USA, 2009. ACM.

[19] Cloud Model 1. http://www.mmm.ucar.edu/people/bryan/cm1/.

[20] Science Clouds. http://www.scienceclouds.org/.

[21] Andrew W. Cooke, Alasdair J. G. Gray, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Roney Cordenonsi, Rob Byrom, Linda Cornwall, Abdeslem Djaoui, Laurence Field, Steve Fisher, Steve Hicks, Jason Leake, Robin Middleton, Antony J. Wilson, Xiaomei Zhu, Norbert Podhorszki, Brian A. Coghlan, Stuart Kenny, David O'Callaghan, and John Ryan. The relational grid monitoring architecture: Mediating information about the grid. *Journal of Grid Computing*, 2(4):323–339, 2004.

[22] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[25] E. Deelman, G. Singh, M. Livny, and al. The cost of doing science on the cloud: the Montage example. In *Supercomputing'08*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] Phillip Dickens and Jeremy Logan. Towards a high performance implementation of MPI-IO on the Lustre file system. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Cloud Computing and Its Applications*, October 2008.

[28] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[29] File System in UserspacE (FUSE). http://fuse.sourceforge.net.

[30] Flexiscale. http://www.flexiscale.com/.

[31] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, nov. 2008.

[32] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[33] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005.

[34] Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[35] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[36] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42:5–18, January 2003.

[37] Jeremy Geelan, Markus Klems, Reuven Cohen, Jeff Kaplan, Douglas Gourlay, Praising Gaw, Damon Edwards, Brian de Haaff, Ben Kepes, Kirill Sheynkman, Omar Sultan, Kevin Hartig, Jan Pritzker, Trevor Doerksen, Thorsten von Eicken, Paul Wallis, Michael Sheehan, Don Dodge, Aaron Ricadela, Bill Martin, Ben Kepes, and Irving W. Berger. Twenty-One Experts Define Cloud Computing. 2009.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

[39] GoGrid. http://www.gogrid.com/.

[40] Google Docs. http://docs.google.fr/.

[41] Google Maps. http://maps.google.fr/.

[42] Grid Analysis and Display System (GrADS). http://www.iges.org/grads/.

[43] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[44] Robert L. Grossman. The case for cloud computing. *IT Professional*, 11(2):23–27, 2009.

[45] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework (second edition). *W3C Recommandation*, pages 240–8491, 2007.

[46] T. Gunarathne, Tak-Lon Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 565 –572, 30 2010-dec. 3 2010.

[47] T. Gunarathne, Tak-Lon Wu, J. Qiu, and G. Fox. Mapreduce in the clouds for science. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 565 –572, 30 2010-dec. 3 2010.

[48] Dan Gunter, Brian Tierney, Keith Jackson, Jason Lee, and Martin Stoufer. Dynamic monitoring of high-performance distributed applications. In *Applications, Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing*, pages 163–170, 2002.

[49] Ibrahim F. Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux Journal*, 2000, November 2000.

[50] A Harutyunyan, P Buncic, T Freeman, and K Keahey. Dynamic virtual alien grid sites on nimbus with cernvm. *Journal of Physics: Conference Series*, 219(7):072036, 2010.

[51] Z. Hill and M. Humphrey. A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 26 –33, oct. 2009.

[52] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[53] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. 2001.

[54] Yvon Jégou, Stephane Lantéri, Julien Leduc, Melab Noredine, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[55] G. Juve, E. Deelman, K. Vahi, and al. Data Sharing Options for Scientific Workflows on Amazon EC2. In *Supercomputing'10*, SC '10, pages 1–9, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Lori M. Kaufman. Data security in the world of cloud computing. *IEEE Security and Privacy*, 7:61–64, July 2009.

[57] Kate Keahey and Tim Freeman. Science clouds: Early experiences in cloud computing for scientific applications. In *CCA '08: Cloud Computing and Its Applications*, Chicago, IL, USA, 2008.

[58] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[59] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D.P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –12, may 2009.

[60] KVM Project. http://www.linux-kvm.org/page/Main_Page.

[61] I. Legrand, H. Newman, R. Voicu, et al. MonALISA: An agent based, dynamic service system to monitor, control and optimize grid based applications. In *Computing for High Energy Physics*, Interlaken, Switzerland, 2004.

[62] Lustre. http://wiki.lustre.org/.

[63] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. *IEEE International Symposium on Cluster Computing and the Grid*, 0:43–52, 2010.

[64] Paul Marshall, Kate Keahey, and Timothy Freeman. Improving utilization of infrastructure clouds. In *IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 205–214. IEEE, 2011.

[65] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[66] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[67] Peter Mell and Tim Grance. The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[68] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[69] Microsoft Office Live. http://www.officelive.com/.

[70] Globus Monitoring and Discovery System. http://www.globus.org/toolkit/mds/.

[71] Jesús Montes, Bogdan Nicolae, Gabriel Antoniu, Alberto Sánchez, and Maria Pérez. Using Global Behavior Modeling to Improve QoS in Cloud Data Storage Services. In *CloudCom '10: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 304–311, Indianapolis, United States, October 2010.

[72] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24, New York, NY, USA, 2009. ACM.

[73] NetCDF (Network Common Data Form). http://www.unidata.ucar.edu/software/netcdf/.

[74] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[75] Bogdan Nicolae. *BlobSeer: Towards efficient data storage management for large-scale, distributed systems*. PhD thesis, Université de Rennes 1, Rennes, France, 2010.

[76] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach. In *Proceedings of the 15th International Euro-Par Conference*, pages 404–416, Delft, Netherlands, 2009.

[77] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 71(2):168–184, 2011.

[78] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In *The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2011)*, San José, CA, United States, June 2011.

[79] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proc. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[80] Open Cloud Computing Interface. http://occi-wg.org/.

[81] Open Grid Forum. http://www.gridforum.org/.

[82] OpenNebula. http://opennebula.org/.

[83] OpenStack. http://www.openstack.org/projects/compute/.

[84] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM.

[85] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pages 247–259. Springer Verlag, 2005.

[86] D.F. Parkhill. *The challenge of the computer utility*. Number p. 246 in The Challenge of the Computer Utility. Addison-Wesley Pub. Co., 1966.

[87] The Eucalyptus Project. http://open.eucaplytus.com.

[88] The Nimbus Project. http://www.nimbusproject.org/.

[89] M. Rahman, R. Ranjan, and R. Buyya. A taxonomy of autonomic application management in grids. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 189 –196, dec. 2010.

[90] Tejaswi Redkar. *Windows Azure Platform*. Apress, 2010.

[91] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[92] S3cmd tools. http://s3tools.org/s3cmd.

[93] Dan Sanderson. *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly Media, Inc., 2009.

[94] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[95] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, May 2010.

[96] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, 2005.

[97] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19:181–187, July 2005.

[98] OpenStack Storage. http://openstack.org/projects/storage/.

[99] Ceph File System. http://ceph.newdream.net/.

[100] HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html.

[101] The Parallel Virtual File System. http://www.pvfs.org/.

[102] Windows Azure Table. http://msdn.microsoft.com/en-us/library/dd179463.aspx.

[103] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience: Research articles. *Concurrency and Computation:Practice and Experience*, 17(2-4):323–356, 2005.

[104] The Boto Python interface to Amazon Web Services. http://code.google.com/p/boto/.

[105] The Django Project. https://www.djangoproject.com/.

[106] The HDF Group. http://www.hdfgroup.org/HDF5/.

[107] Eno Thereska, Brandon Salmon, On Salmon, John Strunk, Matthew Wachs, Michael Abd el malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Saint-Malo)*, pages 3–14. ACM Press, 2006.

[108] Top500. http://www.top500.org/.

[109] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a Cloud definition. *SIGCOMM Computer Communication Review*, 39(1):50–55, 2009.

[110] VisIt. https://wci.llnl.gov/codes/visit/.

[111] Edward Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*, 33(5):18–23, October 2008.

[112] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[113] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[114] Himanshu Yadava. *The Berkeley DB Book*. Apress, 2007.

[115] Serafeim Zanikolas and Rizos Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163–188, 2005.

[116] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. 10.1007/s13174-010-0007-6.

[117] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *5th Intl. Workshop on Peer-to-Peer Systems (IPTPS-2006)*, Santa Barbara, USA, February 2006. Electronic proceedings.

[118] Amazon Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/.