Int. J. Appl. Math. Comput. Sci., 2011, Vol. 21, No. 2, 229–242
DOI: 10.2478/v10006-011-0017-y

amcs

BRINGING INTROSPECTION INTO BLOBSEER: TOWARDS A SELF–ADAPTIVE DISTRIBUTED DATA MANAGEMENT SYSTEM

ALEXANDRA CARPEN-AMARIE [*], ALEXANDRU COSTAN [**], JING CAI [***], GABRIEL ANTONIU [*], LUC BOUGÉ [****]

[*] INRIA Rennes–Bretagne Atlantique/IRISA
Campus Universitaire de Beaulieu, 35042 Rennes, France
e-mail: {alexandra.carpen-amarie,gabriel.antoniu}@inria.fr

[**]Polytechnic University of Bucharest
Department of Computer Science, 313 Spl. Independentei, 060042 Bucharest, Romania
e-mail: Alexandru.Costan@cs.pub.ro

[***]Department of Computer Science
City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong, China
e-mail: Tylor.Cai@student.cityu.edu.hk

[****]Ecole Normale Supérieure de Cachan
Antenne de Bretagne/IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France
e-mail: Luc.Bouge@bretagne.ens-cachan.fr

Introspection is the prerequisite of autonomic behavior, the first step towards performance improvement and resource usage optimization for large-scale distributed systems. In grid environments, the task of observing the application behavior is assigned to monitoring systems. However, most of them are designed to provide general resource information and do not consider specific information for higher-level services. More precisely, in the context of data-intensive applications, a specific introspection layer is required to collect data about the usage of storage resources, data access patterns, etc. This paper discusses the requirements for an introspection layer in a data management system for large-scale distributed infrastructures. We focus on the case of BlobSeer, a large-scale distributed system for storing massive data. The paper explains why and how to enhance BlobSeer with introspective capabilities and proposes a three-layered architecture relying on the MonALISA monitoring framework. We illustrate the autonomic behavior of BlobSeer with a self-configuration component aiming to provide storage elasticity by dynamically scaling the number of data providers. Then we propose a preliminary approach for enabling self-protection for the BlobSeer system, through a malicious client detection component. The introspective architecture has been evaluated on the Grid'5000 testbed, with experiments that prove the feasibility of generating relevant information related to the state and behavior of the system.

Keywords: distributed system, storage management, large-scale system, monitoring, introspection.

1. Introduction

Managing data at a large scale has become a critical requirement in a wide spectrum of research domains, ranging from data-mining to high-energy physics, biology or climate simulations. Grid infrastructures provide typical environments for such data-intensive applications, enabling access to a large number of resources and guaranteeing a predictable quality of service. However, as the exponentially growing data are correlated with an increasing need for fast and reliable data access, data management continues to be a key issue that highly impacts on the performance of applications.

More specifically, storage systems intended for very large scales have to address a series of challenges, such as a scalable architecture, data location transparency, high throughput under concurrent access and the storage of

massive data with fine grain access. Although these requirements are the prerequisites for any efficient data management system, they also imply a high degree of complexity in the configuration and tuning of the system, with possible repercussions on the system's availability and reliability.

Such challenges can be overcome if the system is outfitted with a set of self-management mechanisms that enable autonomic behavior, which can shift the burden of understanding and managing the system state from the human administrator to an automatic decision-making engine. However, self-adaptation is impossible without deep and specific knowledge of the state of both the system and the infrastructure where the system is running on. It heavily relies on introspection mechanisms, which play a crucial role of exposing the system behavior accurately and in real time.

On existing geographically distributed platforms (e.g., 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 (Zanikolas and Sakellariou, 2005).

Even though many grid monitoring applications have been developed to address such general needs (Massie *et al.*, 2004; Gunter *et al.*, 2000), little has been done when it comes to enabling introspection for large-scale distributed data management. This is particularly important in the context of data-intensive applications distributed at a large scale. In such a context, specific parameters related to data storage need to be monitored and analyzed in order to enable self-optimization in terms of resource usage and global performance. Such parameters regard physical data distribution, storage space availability, data access patterns, application-level throughput, etc.

This paper discusses the requirements of a large-scale distributed data management service in terms of self-management. It explains which self-adaptation directions can serve a data management service designed for large-scale infrastructures. Furthermore, it focuses on introspection, identifying the specific ways in which introspection can be used to enable the autonomic behavior of a distributed data storage system.

As a case study, we focus on BlobSeer (Nicolae *et al.*, 2010), a service for sharing massive data at very large scales in a multi-user environment. We propose a three-layered architecture enabling BlobSeer with introspection capabilities. We validate our approach through an implementation based on the generic MonALISA (Legrand *et al.*, 2004) monitoring framework for large-scale distributed services. Moreover, we provide two applications for the introspection layer, targeting self-configuration and self-protection, which take advantage of the introspective features that BlobSeer is equipped with.

The remainder of the paper is organized as follows. Section 2 summarizes existing efforts in the grid monitoring systems field, emphasizing their limitations when it comes to enabling specific introspection requirements. Section 3 explains which self-management directions fit the needs of data management systems. Section 4 provides a brief description of BlobSeer and describes the specific introspection mechanism that we designed and implemented and the data that need to be collected in such a data management system. Section 5 presents the applications of the introspective features of BlobSeer, namely, a self-configuration module dealing with storage elasticity and the preliminary steps towards a self-protection component. In Section 6, we discuss the feasibility and efficiency of our approach, by presenting a visualization tool and a set of experiments realized on the Grid'5000 testbed. Finally, Section 7 draws conclusions and outlines directions for future developments.

2. Related work

The autonomic behavior of large scale distributed systems aims to deal with dynamic adaptation issues by embedding the management of complex systems inside the systems themselves, alleviating the users and administrators from additional tasks. A distributed service, like a storage service, is said to be autonomic if it encapsulates some autonomic behavior (Gurguis and Zeid, 2005) such as self-configuration, self-optimization, self-healing, and self-protection (Kephart and Chess, 2003).

In this context, performance evaluation becomes a critical component of any dynamic system that requires high throughput, scheduling, load balancing or analysis of applications' performance and communications between nodes. In grid environments, previous research has often been limited to using historical information to create models to which various analysis and mining techniques are applied. The results were thereafter used for performing more efficient job mappings on available resources. The autonomic behavior depends on monitoring the distributed system to obtain the data on which decisions are based. Experience with production sites showed that, in large distributed systems with thousands of managed components, the process of identifying the causes of faults in due time by extensive search through the potential root failure injectors proves rather time consuming and difficult. This process may interrupt or obstruct important system services. Several techniques were used to address these issues.

One approach relies on Bayesian Networks (BNs) (Cowell *et al.*, 1999), often used to model systems whose behavior is not fully understood. We investigated some consistent work already done on the

probabilistic management in distributed systems. Hood and Ji (1997) utilize Bayesian networks for proactive detection of abnormal behavior in a distributed system. Steinder and Sethi (2004) apply Bayesian reasoning techniques to perform fault localization in complex communication systems. Ding *et al.* (2004) present probabilistic inference in fault management based on Bayesian networks. However, the Bayesian network paradigm used within all these works does not provide direct mechanisms for modeling the temporal dependencies in dynamic systems (Santos and Young, 1999), which is essential for enhancing autonomic behavior.

Another approach takes time into consideration by identifying the dynamic changes in distributed systems as a discrete nonlinear time series. Previous research work on scalable distributed monitoring for autonomous systems can be broadly classified into two categories: relying on decentralized architectures such as hierarchical aggregation (Van Renesse *et al*., 2003) or the peer-to-peer structure (Albrecht *et al*., 2005) to distribute monitoring workload, and trading off information coverage (Liang *et al*., 2007) or information precision (Jain *et al*., 2007) for lower monitoring cost. In contrast, our research focuses on identifying the relevant parameters for an autonomic introspection layer while relying on the extension and adaptation of some existing monitoring tools for tracking these parameters. The monitoring solution should further meet our needs for non-intrusiveness and minimized monitoring costs.

Exploring correlation patterns among distributed monitoring data sources has been extensively studied in various contexts such as sensor network monitoring (Vuran and Akyildiz, 2006), distributed event tracking (Jain *et al*., 2004), and resource discovery (Cardosa and Chandra, 2008). While the general idea of exploring temporal and spatial correlations is not new, we shall emphasize that applying the idea to distributed information tracking over large-scale networked systems requires nontrivial system analysis and design. In our case, this means discovering dynamic correlation patterns (for some predefined targeted events: node failures, malicious clients intrusions, etc.) among distributed information sources, using light-weight methods instead of assuming a specific probabilistic model, as in wireless sensor networks, for instance.

Although the works mentioned above are able to provide some means of monitoring for singular or aggregate services, they do not dynamically replace the faulty service once a failure has been detected, or take automated actions to optimize the system's overall performance, as our work attempts to do to within a large scale distributed storage system.

3. Self-adaptation for large scale data management systems

A large scale data management platform is a complex system that has to deal with changing rates of concurrent users, the management of huge data spread across hundreds of nodes or with malicious attempts to access or to damage stored data. Therefore, such a system can benefit from a self-adaptation component that enables autonomic behavior. We refine the set of self-adaptation directions that best suit the requirements of data management systems: they match the main self-management properties defined for autonomic systems (Kephart and Chess, 2003; Parashar and Hariri, 2005).

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-optimization is the ability to efficiently allocate and use resources, while dealing with changing workloads. It aims at optimizing the system's performance and increasing data availability.

Self-configuration is the property that addresses the dynamic adaptation of the system's deployment scheme as a response to changing 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.

Self-protection addresses the detection of hostile or intrusive actions directed towards the system's components and enables the system to automatically take appropriate measures to enforce security policies and make itself less vulnerable to subsequent similar attacks.

In order to improve the performance and the efficiency of resource usage in a data sharing system, below we define a set of goals that justify the need for the aforementioned properties.

Monitoring. Constant surveillance of the state of a system and of the events that trigger system reactions is the prerequisite of all the other self-adaptation directions. Thus, the self-awareness property is of utmost importance for providing support for autonomous behavior.

Dynamic dimensioning. The performance of data access primitives is influenced by the number of running nodes of the data sharing system. Moreover, the load of

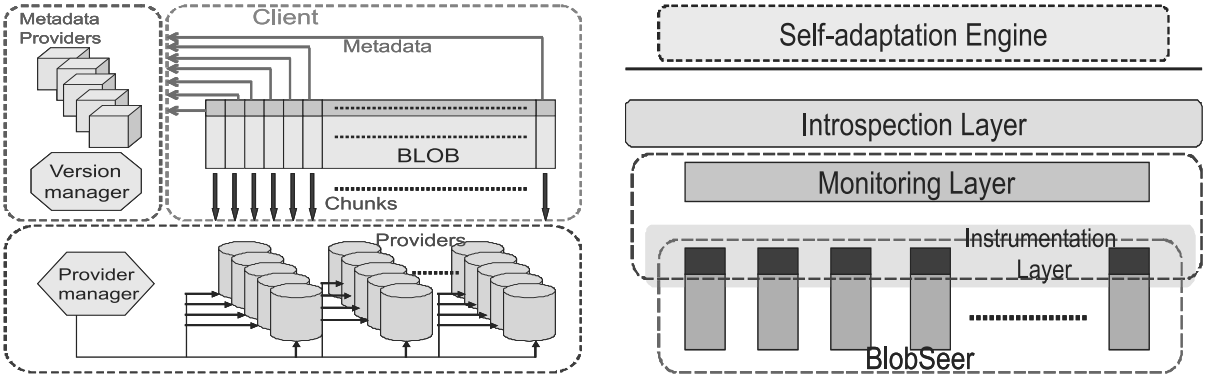


Fig. 1. BlobSeer: architecture of the BlobSeer system (a), architecture of the introspective BlobSeer (b).

each component that stores data is also dependent on the available storage nodes and on their capacity to serve user requests. On the other hand, the workload is often unpredictable, and the deployment of the system on a large number of physical nodes can lead to underused storage nodes when the number of clients is low or the stored data are not large enough. These reasons account for the need to enhance a large-scale storage system with a mechanism that dynamically adjusts the number of deployed storage nodes. This is equivalent to taking advantage of real-time indicators of the state of the system within a self-configuration component that can observe a heavy load or underutilized components.

Malicious clients detection. A data sharing system distributed on a large number of nodes can fit the needs of applications that generate important amounts of data only if it can provide a degree of security for the stored information. For this reason, the system has to be able to recognize malicious requests generated by unauthorized users and to block illegal attempts to inject or to modify data. Therefore, a self-protection component that enforces these requirements has to be integrated into the system.

4. Towards an introspective BlobSeer

BlobSeer is a data sharing system which addresses the problem of efficiently storing massive, unstructured data blocks called *binary large objects* (referred to as BLOBs further in this paper), in large-scale, distributed environments. The BLOBs are fragmented into small, equally-sized *chunks*. BlobSeer provides efficient fine-grained access to the chunks belonging to each BLOB, as well as the possibility to modify them, in distributed, multi-user environments.

4.1. Architecture. The architecture of BlobSeer (cf. Fig. 1(a)) includes multiple distributed entities. *Clients* initiate all BLOB operations: CREATE, READ, WRITE and APPEND. There can be many concurrent clients accessing the same BLOB or different BLOBs at the same time. The support for concurrent operations is enhanced by storing the chunks belonging to the same BLOB on multiple *data providers*. The metadata associated with each BLOB are hosted on other components, called *metadata providers*. BlobSeer provides versioning support, so as to prevent chunks from being overwritten and to be able to handle highly-concurrent WRITE and APPEND operations. For each of them, only a patch composed of the range of written chunks is added to the system. Finally, the system comprises two more entities: the *version manager*, which deals with the serialization of the concurrent WRITE/APPEND requests and with the assignment of version numbers for each new WRITE/APPEND operation, and the *provider manager*, which keeps track of all storage providers in the system.

A typical setting of the BlobSeer system involves the deployment of a few hundreds data providers, storing BLOBs of the order of the TB. A 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 access to the stored data, reaching a throughput of a 6.7 GB/s aggregated bandwidth for a configuration with 60 metadata providers, 90 data providers and 360 concurrent writers, as explained by Nicolae *et al*. (2009).

4.2. Introspection mechanisms on top of BlobSeer. We enhanced BlobSeer with introspection capabilities in order to enable this data sharing platform with autonomic behavior. Carpen-Amarie *et al*. (2010) present a three-layered architecture designed to identify and generate relevant information related to the state and the behavior of the system (Fig. 1(b)). Such information is then expected to serve as input to a higher-level *self-adaptation* engine. These data are yielded by (i) an *introspection* layer, which

processes the raw data collected by (ii) a *monitoring* layer. The lowest layer is represented by the (iii) *instrumentation* code that enables BlobSeer to send monitoring data to the upper layers.

4.2.1. Introspection: Relevant data. The self-adaptation engine can only be effective if it receives accurate data from the *introspection layer*. The latter generates data ranging from general information about the running nodes to specific data regarding the stored BLOBs and their structure.

General information. These data are essentially concerned with the physical resources of the nodes that act as data 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, by means of aggregated data. For instance, the occupied and available storage space at each single provider play a crucial role in deciding whether or not additional providers are needed.

Individual BLOB-related data. The most significant information for a single BLOB is its access pattern, i.e., the way the chunks and the versions are accessed through READ and WRITE operations. The basic data are the number of read access instances for each chunk that the BLOB version consists of, and the number of WRITE operations performed on the BLOB for each chunk. These data facilitate the identification of BLOB regions comprising chunks with a similar number of access instances, information that can influence the adopted replication strategy.

Global state. Even though the details within each BLOB are made available to the provider-allocation algorithms implemented by the provider manager, it is essential to have an overview of the whole data stored in the BlobSeer system, from a higher-level point of view. Some of the key data at this global level are the total number of access instances associated with each provider. This is a measure of the load of each of them and can directly influence the selection of the providers that will be allocated new chunks, depending on their deviation from the average load within the system.

4.2.2. Monitoring: Data collection mechanisms. The input for the introspective layer consists of raw data that are extracted from the running nodes of BlobSeer, collected and then stored, a set of operations realized within the *monitoring layer*. Therefore, it can rely on a monitoring system designed for large-scale environments that implements these features. Such a monitoring framework has to be both scalable and extensible, so as to be able to deal with the huge number of events generated by a large-scale data management system, as well as to accommodate system-specific monitoring information and to offer a flexible storage schema for the collected data.

Monitoring framework: MonALISA. The Global Grid Forum (GGF, 2010) proposed a Grid Monitoring Architecture (GMA) (Tierney *et al*., 2002), which defines the components needed by a scalable and flexible grid monitoring system: producers, consumers, and a directory service. A wide variety of grid monitoring systems (Zanikolas and Sakellariou, 2005), such as Ganglia (Massie *et al*., 2004), RGMA (Cooke *et al*., 2004), GridICE (Andreozzia *et al*., 2005), comply with this architecture.

Among them, we selected MonALISA (*Monitoring Agents in a Large Integrated Services Architecture*) (Legrand *et al*., 2004) for our data monitoring tasks, as it is a general-purpose, flexible framework, which provides the necessary tools for collecting and processing monitoring information in large-scale distributed systems. Moreover, it is an easily extensible system, which allows the definition and processing of user-specific data, by means of an API for dynamically-loadable modules. MonALISA is currently used to monitor large high-energy physics facilities. It is deployed on over 300 sites belonging to several experiments, such as CMS or ALICE (ALICE, 2010).

In BlobSeer, the main challenge the monitoring layer has to cope with is the large number of data provider nodes and therefore the huge number of BLOB chunks, versions and huge BLOB sizes. Furthermore, it has to deal with hundreds of clients that concurrently access various parts of the stored BLOBs, as they generate a piece of monitoring information for each chunk accessed on each provider. MonALISA is suitable for this task as it is a system designed for large-scale environments and proved to be both scalable and reliable.

Instrumenting BlobSeer. The data generated by the *instrumentation layer* are relayed by the monitoring system and finally fed to the introspection layer. The instrumentation layer is implemented as a component of the monitoring layer. The MonALISA framework provides a library called ApMon that can be used to send the monitoring data to the MonALISA services. At the providers, the instrumentation code consists of listeners located on each of them, which report to the monitoring system each time a chunk is written or read. The monitoring information from the version manager is collected using a parser that monitors the events recorded in the logs. The state of the physical resources on each node is monitored through an ApMon thread that periodically sends data to the monitoring service.

5. Introducing self-adaptation for BlobSeer

To introduce autonomic behavior in BlobSeer, we investigated two directions. The first approach aims at enhancing BlobSeer with self-configuration capabilities, as a means to support storage elasticity through dynamic deployment of data providers. The second direction addresses the self-protection of BlobSeer from malicious clients by detecting and reacting to potential threats in realtime based on the information yielded by the introspection layer. In this section, we detail these two approaches.

5.1. Self-configuration through dynamic data providers deployment. Dynamic dimensioning is a means to achieve the self-configuration of BlobSeer, by enabling the data providers to scale up and down depending on the detected system needs. The component we designed adapts the storage system to the environment by contracting and expanding the pool of data providers based on the system's load. The key idea of the dynamic data providers deployment component is the automatic decision that has to be made on how many resources the system needs to operate normally while keeping the resources utilization down to a minimum. This problem is addressed by using a test-decided heuristic based on the monitoring data. The system maintains two pools of providers:

Active Pool of Providers (APP) a pool of providers that are currently on and are actively used by the BlobSeer infrastructure.

Backup Pool of Providers (BPP) a pool of providers that are currently off, waiting in stand-by to be activated in order to be used.

The goal is to dynamically switch providers from one pool to another when certain conditions are met, in order to optimize resource usage. Instead of reserving a large number of nodes which eventually are not effectively used, the system only relies on the APP and self-adapts its execution using the BPP.

5.1.1. Architectural overview. The dynamic deployment decision is based on retrieving the monitoring data and computing a score that evaluates the status of each provider. The monitoring data are retrieved from two different sources, each one with specific metrics: BlobSeer-related data and physical resources information. These data are stored and processed using a monitoring repository. Based on the real-time monitoring information, the decision algorithm computes a heuristic score. Its value determines the decision of removing or adding a node to the active pool of providers.

In order to take the corresponding action based on the obtained result, the application needs to get a list of available nodes (data providers) from the provider manager which can be turned on or off, depending on the decision taken. This part is also responsible for notifying the BlobSeer system, specifically the provider manager, of the changes made in the system.

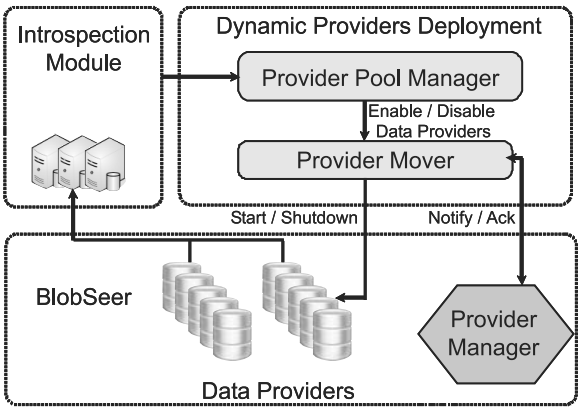


Fig. 2. Architectural overview of the dynamic deployment module.

The main actors of the dynamic deployment service are the decision taking component (the provider pool manager) and the decision enforcement component (the provider mover), as depicted in Fig. 2. The provider pool manager analyzes the monitoring information and using some configurable policies it makes the decision about either enabling or disabling a set of data providers. The provider mover is responsible for putting this decision into practice by moving a provider from the active pool of providers to the backup pool of providers or vice-versa, depending on what commands it receives from the provider pool manager.

The interaction with BlobSeer's provider manager is represented by requests for the list of the active data providers running in the system at a specific moment in time. The provider pool manager reads the coordinates of the provider manager and contacts it to obtain a list of tuples `(host, port)` that point to the nodes where data providers are active. The provider mover also manages the two pools, APP and BPP, and the Providers' migration between them. The provider mover notifies the provider manager of a change in the APP. If the notification fails, the provider mover does not retry it, relying on the watchdog facility implemented in BlobSeer, which scans the entire list of providers to track the active providers. Finally, the provider mover communicates directly with the data providers and issues the *start* or *shutdown* commands through which a provider is moved from BPP to APP or from APP to BPP, respectively.

The sequence diagram depicted in Fig. 3 illustrates the flow of actions within the dynamic deployment module. The monitoring data are retrieved continuously, as a separate process by the monitoring module, and are stored into a monitoring repository. The provider pool manager connects to the provider manager to get the list of active

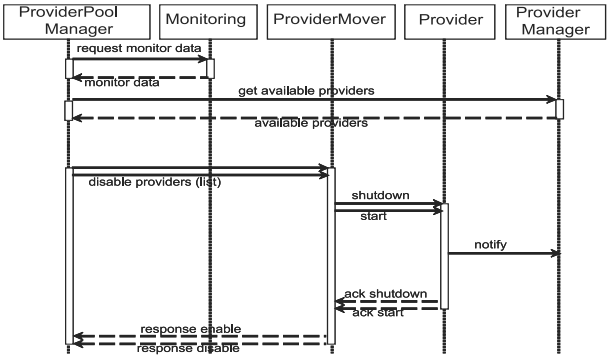


Fig. 3. Sequence diagram of dynamic providers deployment.

providers. Once these data are obtained, the pool manager starts computing a score for each provider. Based on a configuration file specifying the targeted scenarios and the heuristic used, a decision is taken and communicated to the provider mover. This, in turn, calls the scripts that start or stop a particular provider.

5.1.2. Heuristic providers evaluation. The scoring algorithm provides a method to detect which providers should be moved from the APP to the BPP. The factors to be taken into consideration and tracked using the introspection layer can be divided into two subcategories: *physical factors* (depending on the physical node that runs the data provider, e.g., the free disk space, the average bandwidth usage, the CPU load or the system uptime) and *BlobSeer factors* (metrics referring to the BlobSeer behavior, e.g., the number of read/write access instances per time unit, the size of stored chunks and BLOBs, the replication degree).

We illustrate this approach with a common scenario identified by the dynamic provider deployment module and treated accordingly by stopping the unnecessary providers. In this case, if the introspection layer detects that on one provider the free disk space is above the 70% threshold, the replication factor for the stored chunks is greater than 1, with a small read and write access rate (e.g., less than one access per hour), it decides to shut down the provider. All the values referred to above are adjustable through a configuration file. The current values were chosen based on a set of performance evaluation experiments aiming to identify the tradeoff between the costs of shutting down one provider and moving its data to another one, and the benefits of using fewer resources. The scenario illustrates the case of a provider with extra disk space available, which is not used by clients. Considering all the stored data are also replicated on other providers, it is reasonable to shut down this provider in order to efficiently use the available resources. The shut down decision is only made when the operation's costs are smaller and there are available nodes where the data can be transferred to preserve the replication factors.

Algorithm 1 Scaling down data providers.

```
1:  procedure SCALING_DOWN(dataProvidersList)
2:      for all dataProvider in dataProvidersList do
3:          GET_MONITORING_DATA(dataProvider)
4:          S ← COMPUTE_SCORE(dataProvider)
5:          if S < scoreThreshold then
6:              keep dataProvider in APP
7:          else
8:              if dataReplicationDegree > replicationThreshold then
9:                  move dataProvider to BPP
10:                 availableProviders ← retrieve available providers from the provider manager
11:                 transfer data to availableProviders
12:                 update metadata
13:                 SHUT_DOWN(dataProvider)
14:             else
15:                 keep dataProvider in APP
16:             end if
17:         end if
18:     end for
19: end procedure
```

The self-configuration engine is not limited to detecting this type of scenarios. Several other patterns are identifiable using a simple specification mechanism. The conditions making up the scenarios are modeled as factors used to compute a score for each provider. The heuristic used in the score computation is based on weight factors using the following formula:

$$S = \sum_{i=1}^{n} wft_i \cdot wcf_i, \tag{1}$$

where wft_i represents the weight of the factor i from the total score and wcf_i represents the weight of the true condition from the factor i. With this notation, the pseudocode for scaling down data providers is presented in Algorithm 1.

5.2. Self-protection through malicious client detection. Detecting malicious clients is the first step towards enabling self-protection for the BlobSeer system. Such a feature has to take into account several types of security threats and to react when such attacks occur.

In this section, we propose a simple malicious client detection mechanism that focuses on protocol breaches within BlobSeer, as this is a critical vulnerability of a data management system that enables the clients to directly access the data storage nodes in order to provide very efficient data transfers. The goal of the detection component is to identify the known forms of protocol misuse, and

Algorithm 2 Data writing step.

1: **procedure** WRITE_DATA($buffer$, $offset$, $size$)
2: $wId \leftarrow$ generate unique write id
3: $noCh \leftarrow \lceil \text{size}/chSize \rceil$
4: $P \leftarrow$ **get** $noCh$ providers **from** provider manager
5: $D \leftarrow \emptyset$
6: **for all** $0 \leq i < noCh$ **in parallel do**
7: $chId \leftarrow$ generate unique chunk id
8: $chOffset \leftarrow chSize \times i$
9: **store** $buffer[chOffset\ ..\ chOffset + chSize]$ as chunk $(chId, wId)$ on provider $P[i]$
10: $D \leftarrow D \cup \{(chId, wId, i, chSize)\}$
11: $P_{\text{global}} \leftarrow P_{\text{global}} \cup \{(chId, P[i])\}$
12: **end for**
13: **end procedure**

thus to help the system to maintain the stored data in a consistent state.

5.2.1. Protocol breach scenarios for BlobSeer. A malicious user can try to compromise the system by deliberately breaking the data insertion protocols. This kind of behavior is a starting point for DoS attacks, in which the user attempts to overload the system through large numbers of malformed or incomplete requests. To cope with this security risk, specific mechanisms have to be developed to quickly detect the illegal access instances and isolate the user that initiated them.

The most vulnerable data access operation is writing data into BlobSeer, as it gives a malicious user not only the opportunity to overload the system and to increase its response time, but also the means to make corrupted data available.

The WRITE operation imposes a strict protocol on the user that wants to correctly insert data into the system. We consider the typical case of WRITE operations in BlobSeer, that is, when a user attempts to write a continuous range of chunks to a specific BLOB. For simplicity, we can assume that the WRITE operation consists of two independent phases that have to be executed consecutively. These two steps can be summarized as follows (a full description of the data access primitives in BlobSeer can be found in the work of Nicolae *et al*. (2010)):

Data writing step. A simplified description of this operation is provided in Algorithm 2. We assume the size of data to be written is a multiple of a predefined chunk size, denoted by *chSize*, as this is often the case in BlobSeer. The input parameters of this step are the data to be written as a string *buffer*, the *offset* within the BLOB where the data have to be inserted and the *size* of the sequence.

The client connects to the provider manager and requests a list of data providers, P, which can host the chunks to be written. Then, the chunks are sent in parallel to the data providers, together with a unique identifier, *chId*, and the identifier of the WRITE operation, *wId*. Upon successful completion of this step, the information associated with all the written chunks will be stored in a chunk descriptor map denoted by D. Additionally, the providers that hold each *chId* are stored in P_{global}, a container where the addresses of all the chunks in the system are saved.

Algorithm 3 Data publication step.

1: **procedure** PUBLISH_DATA($offset$, $size$, D, wId)
2: $writeInfo \leftarrow$ **invoke remotely** on version manager ASSIGN_VERSION($offset$, $size$, wId)
3: BUILD_METADATA($writeInfo$, D)
4: **invoke remotely** on version manager COMPLETE_WRITE ($writeInfo$)
5: **end procedure**

Data publication step. It is represented by the creation of the metadata associated with the written data and the publication of the written chunk range as a new version, as described in Algorithm 3.

First, the client asks the version manager for a new version for its chunk list, and then it proceeds to the creation of metadata, starting from the chunk descriptor map D generated in the first step. The WRITE IS operation finalized after the client successfully invokes the COMPLETE_WRITE procedure on the version manager, which in turn is responsible for publishing the new version of the BLOB.

A correct WRITE operation is defined as successful completion of the aforementioned steps, with the constraint that the published information concerning the written chunk range be consistent with the actual data sent to the data providers, that is, the values of D and wId that are sent to the version manager correspond to chunks that have been written on data providers. As a consequence, there are two 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. This kind of protocol breach can be developed into a Denial of Service (DoS) attack, aiming to overload of one or more data providers.

Publication of inconsistent data. The attack that corresponds to this situation aims to disrupt the computations that use data stored by the BLOBs. As an example, a user may attempt to compromise the system by making data that do not actually exist available. Therefore, an ap-

plication can start reading and processing the data without being aware that the metadata contain fake references. Hence the computation would be compromised and the application forced to restart the processing.

5.2.2. Detection mechanism. Enabling self-protection in BlobSeer relies on coupling a malicious-client detection module with the introspection layer. On the one hand, such a module has to identify the malicious activities that attempt to compromise the system and to isolate users that initialize them. On the other hand, it should not interfere with BlobSeer operations, so as to preserve the efficient data access instances for which BlobSeer is optimized. The introspection layer processes information monitored independently of the interactions between the user and the system, and thus it is an ideal candidate to provide input data for a malicious client detection module.

We implemented a detection module that addresses the protocol-breach attacks and generates *blacklists* with the users that attempt them. Its input data are provided as a history of the users' actions by the introspection layer, which constantly monitors the real-time data access and updates the history. The user history stores the following types of monitoring parameters:

Data generated by the data providers. The monitoring information collected from the data providers consists of tuples that aggregate the information about the stored data chunks. The data corresponding to a new chunk written in the system are defined as a tuple denoted by $(cId, wId, noCh, chSize, ts)$, where cId is the client identifier, wId is the write identifier generated in the data writing step and ts is the timestamp attached by the monitoring system when the data are recorded. Note that for each wId there can be several records in the user history (with different timestamps), as not all the chunk writes are recorded by the monitoring system at the same time.

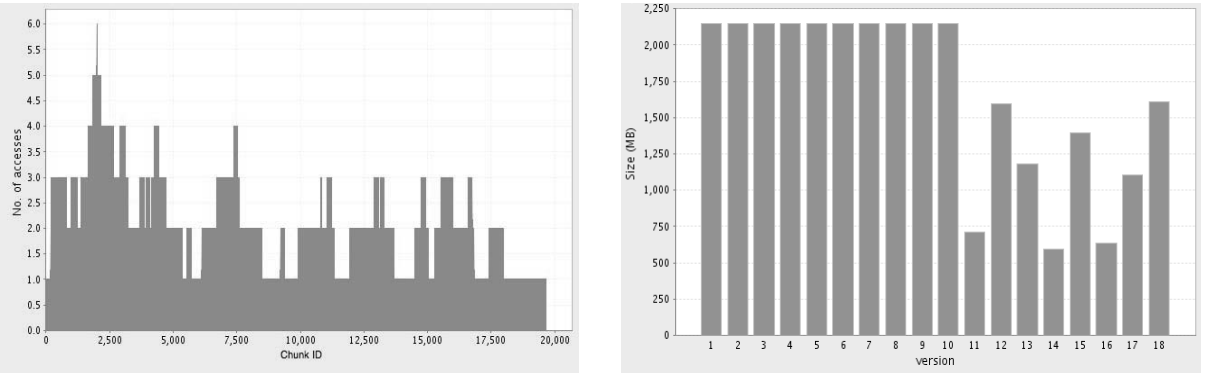
Data obtained from the version manager. The introspection system records each new version published by the version manager in the form of tuples defined as $(cId, wId, v, offset, size, ts)$, where cId is the client identifier, wId is the same write identifier used for the data writing step, v is the new published version, $offset$ and $size$ identify the chunk range written into the system, and ts is the timestamp assigned by the monitoring system.

The detection module comprises two components, each of them dealing with a specific type of protocol breach. The detection mechanism for inconsistent data publication is presented in Algorithm 4. The DETECT_ILLEGAL_PUBLISH procedure is executed periodically and each time it inspects the most recent monitoring data recorded by the introspection module. The procedure searches for published versions that have no corresponding written data chunks or the written range of chunks does not match the information published. Each published write is matched against the set of chunk writes that occurred in a predefined time window, denoted by *windowSize*, surrounding its timestamp. If no chunk writes are found with the same client identifier and write id, or if the total size of the written chunks does not match the published size, the client is added to a global blacklist BL. Once blacklisted, a client is also associated with a score, which can be computed according to the type of illegal action. For example, if no chunks are written, the UPDATE_SCORE procedure computes a score proportional to the write size declared by the publication step.

Algorithm 4 Malicious clients detection.

```
1:  BL ← ∅
2:  lastTsChecked = 0
3:  procedure DETECT_ILLEGAL_PUBLISH
4:      maxTs = getCurentTime() − windowSize
5:      PW ← get list of published writes such that ts > lastTsChecked and ts ≤ maxTs
6:      DW ← get list of data writes such that ts > lastTsChecked − windowSize
7:      lastTsChecked ← max(ts) from PW
8:      for p ∈ PW, p = (cId, wId, offset, size, v) do
9:          if ∄d ∈ DW, d = (cId_d, wId_d, noCh_d, chSize_d, ts_d) such that cId_d = cId, wId_d = wId then
10:             BL ← UPDATE_SCORE(BL, cId, p)
11:         else
12:             if size ≠ Σ_{d∈DW} noCh_d × chSize_d then
13:                 BL← UPDATE_SCORE(BL, cId, p)
14:             end if
15:         end if
16:     end for
17: end procedure
```

The goal of the detection mechanism is to keep track of the malicious users and to feed this information back into the BlobSeer system, so as to enable it to react when receiving new requests from the users identified as malicious. The malicious users can be made available to the provider manager as a *blacklist* where each user's score shows the amount of fake data that the user introduced into the BlobSeer system. The provider manager implements the allocation strategy that assigns providers for each user WRITE operation. Being aware of the *blacklist*, the provider manager can decide to block the malicious users by not granting the providers when they want to write again into the system. The behavior of the provider manager can be further refined by taking into account the score associated with each client. In this case, there are several other constraints that can be enforced on the users, such as a decreased bandwidth for their WRITE operations, a waiting time imposed before being assigned the



(a) (b)

Fig. 4. Visualization for BlobSeer-specific data: number of WRITE access instances on each chunk of a BLOB (each chunk is identified by its position within the BLOB) (a), size of all the stored versions of a BLOB (b).

necessary list of providers or a size limit for the data written.

6. Experimental evaluation

We evaluated the feasibility of gathering and interpreting the BlobSeer-specific data needed as input data for the different self-optimizing directions. Our aim was to create an introspection layer on top of the monitoring system, able to process the raw data collected from BlobSeer and to extract significant information regarding the state and behavior of the system. We performed a series of experiments that evaluate the introspection layer and also provide some preliminary results concerning the introduction of self-protection capabilities in BlobSeer. The experiments were conducted on the Grid'5000 (Bolze *et al.*, 2006) testbed, a large-scale experimental grid platform that covers nine sites geographically distributed across France.

6.1. Visualization tool for BlobSeer-specific data. We implemented a visualization tool that can provide a graphical representation of the most important parameters yielded by the introspection layer.

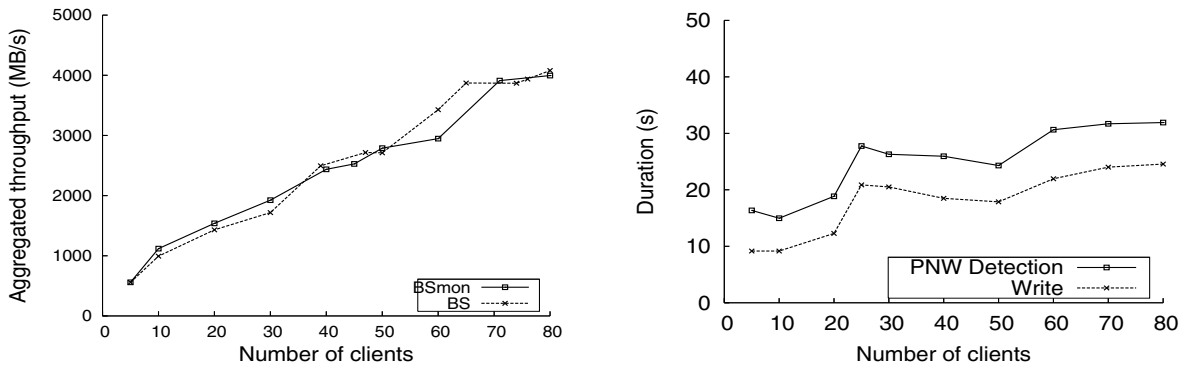
We show the outcome of the introspection layer through an evaluation performed on 127 nodes belonging to a Grid'5000 cluster in Rennes. The nodes are equipped with x86_64 CPUs and at least 4 GB of RAM. They are interconnected through a Gigabit Ethernet network. We deployed each BlobSeer entity on a dedicated 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 a MonALISA repository. The repository is the location where the data were stored and made available to the introspection layer.

In this experiment, we used 10 BLOBs, each of them having the 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 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 significant information that the introspection layer has to be aware of. It can be obtained by computing the number of READ/WRITE access instances. 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 access instances (Fig. 4(a)). The other one refers to the number of READ or WRITE operations performed on each provider, allowing classification of the providers according to the pressure of the concurrent access they have to withstand.

Size of all the stored versions of a BLOB. The differences between the versions of the same BLOB are presented in Fig. 4(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 access instances for each version, can be used to identify ver-



(a) (b)

Fig. 5. Performance evaluations: aggregated throughput of the WRITE operation for BlobSeer (BS) and for BlobSeer with the monitoring support enabled (BSMON) (a), WRITE duration and the detection delay when concurrent clients that publish data without writing them (PNW) access the BlobSeer system (b).

sions that correspond to a small amount of data and are seldom accessed. Such observations are necessary for a self-optimization component that handles the replication degree of each version.

6.2. Impact of the introspection architecture on BlobSeer data access performance. This experiment is designed to evaluate the impact of using the BlobSeer system in conjunction with the introspection architecture. The introspective layer collects data from BlobSeer without disrupting the interactions between its components, and thus no constraint is enforced on the user's access to the BlobSeer entities. In this way, the throughput of the BlobSeer system is not influenced by the detection module. The only downside of such a system is the intrusiveness of the instrumentation layer that runs at the level of the BlobSeer components and is susceptible of decreasing 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, which enables the system to store massive amounts of data that can reach the order of the TB. It consists 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. The repository that gathers all the monitored parameters is located outside Grid'5000, as well as the detection module that interacts only with the repository's database. Each entity is deployed on a dedicated physical machine.

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 with 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 clients deployed. Figure 5(a) 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. Since the introspective 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 introspective component performs well even when the number of generated monitoring parameters reaches $10,000$, as is the case when testing it with more than 80 clients.

6.3. Malicious clients detection. We aim to explore the first step towards a self-protecting BlobSeer system, by building a component that can detect illegal actions and prevent malicious users from damaging the stored data. To reach this goal, the detection mechanism for the malicious users has to deliver an accurate image of the users' interaction with BlobSeer. Moreover, it has to expose the illegal operations as fast as possible, so as to limit the size of data illegally injected into the system and to prevent the malicious users from carrying on the harmful access. We define the detection delay as the duration of the detection phase after the end of the client's operations. We use the detection delay as a measure of the performance of the detection module.

The aim of this experiment is to analyze the perfor-

mance of the detection module when the system is accessed by multiple concurrent malicious clients that publish data without actually writing them. This access pattern corresponds to a scenario where a number of clients access a reputation-based data storage service. Each client can increase its reputation by sharing a large amount of data with the other users of the system. To achieve this goal, a malicious client may pretend to share huge data, while it only skips the data writing phase of the WRITE operation and publishes nonexistent data.

The deployment settings are identical with the previous experiment. We want to assess the behavior of the system under illegal concurrent access. Thus we deploy only malicious clients, repeating the test with an increasing number of clients, ranging from 5 to 80. We measure both the duration of the WRITE operation of the client and the delay between the beginning of the WRITE and the detection of the client that initiated it as being malicious. All the clients start writing at the same time, thus having the same start time. For each point in the chart, we compute the average duration between all the clients deployed for that run. The results obtained in Fig. 5(b) show that the delay between the end of the write operation and the detection of the malicious clients remains constant as the number of clients increases. This is a measure of the scalability of our approach, showing that the detection process is able to cope with a large number of concurrent clients and to deliver results fast enough to allow the system to block the attackers while sustaining the same level of performance.

7. Conclusions and future work

This paper addresses the challenges raised by the introduction of introspection into a data management system for large-scale, distributed infrastructures. Such a feature aims at exposing general and service-specific data to a higher-level layer, in order to enable the system to evolve towards an autonomic behavior. We propose a layered architecture built on top of the BlobSeer data management system, a service dedicated to large-scale sharing of massive data. The goal of this architecture is to generate a set of specific data that can serve as input for a self-adaptive engine.

We also proposed a dynamic dimensioning module and a malicious clients detection component that rely on data yielded by the introspection layer. By reacting in real time to changes in the state of the system, they represent the first step towards enhancing this system with self-configuration and self-protection capabilities.

To build the monitoring layer, we relied on the MonALISA general purpose, large-scale monitoring framework, for its versatility and extensibility. Our experiments showed that it was able to scale with the number of BlobSeer providers and to cope with the huge amount of monitoring data generated by a large number of clients. Moreover, it allowed us to define and collect BlobSeer-specific data, as well as to visualize graphical representations associated with the various high-level data extracted.

The next step will consist in equipping BlobSeer with other self-adaptive components in order to optimize the system's performance and resource usage. For example, by allowing the provider manager to rely on introspection data, this engine will help improving the storage resource allocation strategies. Besides, it can also provide information based on which adaptive data replication strategies can be implemented. Together, such features will enable autonomic behavior of the BlobSeer data management platform.

Acknowledgment

Experiments presented in this paper 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 `http://www.grid5000.org/`).

References

Albrecht, J., Oppenheimer, D., Vahdat, A. and Patterson, D.A. (2005). Design and implementation tradeoffs for wide-area resource discovery, *Proceedings of 14th IEEE Symposium on High Performance, Research Triangle Park, NC, USA*, pp. 113–124.

ALICE (2010). The MonALISA Repository for ALICE, `http://pcalimonitor.cern.ch/map.jsp`.

Andreozzi, S., De Bortoli, N., Fantinel, S., Ghiselli, A., Rubini, G.L., Tortone, G. and Vistoli, M. C. (2005). GridICE: A monitoring service for grid systems, *Future Generation Computer Systems* **21**(4): 559–571.

Bolze, R., Cappello, F., Caron, E., Dayd, M.J., Desprez, F., Jeannot, E., Jgou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Qutier, B., Richard, O., Talbi, E., and Touche, I. (2006). Grid'5000: A large scale and highly reconfigurable experimental grid testbed, *International Journal of High Performance Computing Applications* **20**(4): 481–494.

Cardosa, M. and Chandra, A. (2008). Resource bundles: Using aggregation for statistical wide-area resource discovery and allocation, *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008), Beijing, China*, pp. 760–768.

Carpen-Amarie, A., Cai, J., Costan, A., Antoniu, G. and Bougé, L. (2010). Bringing introspection into the BlobSeer data-management system using the MonALISA distributed monitoring framework, *1st International Workshop on Autonomic Distributed Systems (ADiS 2010), Cracow, Poland*, pp. 508–513.

Cooke, A., Gray, A., Nutt, W., Magowan, J., Oevers, M., Taylor, P., Cordenonsi, R., Byrom, R., Cornwall, L., Djaoui, A., Field, L., Fisher, S., Hicks, S., Leake, J., Middleton, R., Wilson, A., Zhu, X., Podhorszki, N., Coghlan, B., Kenny, S., Callaghan, D.O. and Ryan, J. (2004). The relational grid monitoring architecture: Mediating information about the grid, *Journal of Grid Computing* **2**(4): 323–339.

Cowell, R.G., Dawid, A.P., Lauritzen, S.L. and Spiegelhalter, D.J. (1999). *Probabilistic Networks and Expert Systems*, Springer-Verlag, New York, NY.

Ding, J., Krämer, B.J., Bai, Y. and Chen, H. (2004). Probabilistic inference for network management, *in* M.M. Freie, P. Chemovil, P. Lorenz and A. Gravey (Eds.), *Universal Multiservice Networks*, Lecture Notes in Computer Science, Vol. 3262, Springer, Berlin/Heidelberg, pp. 498–507.

GGF (2010). The Global Grid Forum, `http://www.ggf.org/`.

Gunter, D., Tierney, B., Crowley, B., Holding, M. and Lee, J. (2000). Netlogger: A toolkit for distributed system performance analysis, *MASCOTS '00: Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, CA, USA*, pp. 267–273.

Gurguis, S. and Zeid, A. (2005). Towards autonomic web services: Achieving self-healing using web services, *DEAS05: Proceedings of Design and Evolution of Autonomic Application Software Conference, St. Louis, MO, USA*, pp. 1–5.

Hood, C. and Ji, C. (1997). Automated proactive anomaly detection, *Proceedings of the IEEE International Conference of Network Management (IM97), San Diego, CA, USA*, pp. 688–699.

Jain, A., Chang, E.Y. and Wang, Y.-F. (2004). Adaptive stream resource management using Kalman filters, *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France*, pp. 11–22.

Jain, N., Kit, D., Mahajan, P., Yalagandula, P., Dahlin, M. and Zhang, Y. (2007). STAR: self-tuning aggregation for scalable monitoring, *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases, Vienna, Austria*, pp. 962–973.

Kephart, J.O. and Chess, D.M. (2003). The vision of autonomic computing, *Computer* **36**(1): 41–50.

Legrand, I., Newman, H., Voicu, R., Cirstoiu, C., Grigoras, C., Dobre, C., Muraru, A., Costan, A., Dediu, M. and Stratan, C. MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems, *Computer Physics Communications* **180**(12): 2472–2498.

Liang, J., Gu, X. and Nahrstedt, K. (2007). Self-configuring information management for large-scale service overlays, *INFOCOM 2007: 26th IEEE International Conference on Computer Communications/Joint Conference of the IEEE Computer and Communications Societies, Anchorage, AK, USA*, pp. 472–480.

Massie, M., Chun, B. and Culler, D. (2004). The Ganglia distributed monitoring system: Design, implementation, and experience, *Parallel Computing* **30**(7): 817–840.

Nicolae, B., Antoniu, G. and Bougé, L. (2009). Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach, *Proceedings of the 15th International Euro-Par Conference, Delft, The Netherlands*, pp. 404–416.

Nicolae, B., Antoniu, G., Bougé, L., Moise, D. and Carpen-Amarie, A. (2010). BlobSeer: Next generation data management for large scale infrastructures, *Journal of Parallel and Distributed Computing* **71**(2): 168–184.

Parashar, M. and Hariri, S. (2005). Autonomic computing: An overview, *in* J.-P. Banâtre, P. Fradet, I. -L. Giavitto and O. Michel (Eds.), *Unconventional Programming Paradigms*, Lecture Notes in Computer Science, Vol. 3566, Springer Berlin/Heidelberg, pp. 247–259.

Santos, Jr., E. and Young, J. D. (1999). Probabilistic temporal networks: A unified framework for reasoning with time and uncertainty, *International Journal of Approximate Reasoning* **20**(3): 263–291.

Steinder, M. and Sethi, A. S. (2004). Probabilistic fault localization in communication systems using belief networks, *IEEE/ACM Transactions on Networking* **12**(5): 809–822.

Tierney, B., Aydt, R. and Gunter, D. (2002). A grid monitoring architecture, *Grid Working Draft GWD-PERF-16-3* `http://www.gridforum.org/`.

Van Renesse, R., Birman, K.P. and Vogels, W. (2003). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, *ACM Transactions on Computer Systems* **21**(2): 164–206.

Vuran, M.C. and Akyildiz, I.F. (2006). Spatial correlation-based collaborative medium access control in wireless sensor networks, *IEEE/ACM Transactions on Networking* **14**(2): 316–329.

Zanikolas, S. and Sakellariou, R. (2005). A taxonomy of grid monitoring systems, *Future Generation Computing Systems* **21**(1): 163–188.

Alexandra Carpen-Amarie received her engineering degree in 2008 from the Computer Science Department of Bucharest Polytechnic University, Romania. She is currently a Ph.D. student at Ecole Normale Superieure de Cachan, Antenne de Bretagne, France, working in the KerData Team at the Rennes–Bretagne Atlantique research center of the French National Institute of Computer Science and Control (INRIA). Her research interests include large-scale distributed data storage, cloud computing and monitoring in distributed systems.

Alexandru Costan is a postdoctoral researcher within the KerData team at the Rennes–Bretagne Atlantique research center of the French National Institute of Computer Science and Control (INRIA). He obtained his Ph.D. in 2011 from the Polytechnic University of Bucharest. His research interests include data management for large scale distributed infrastructures and cloud data services, monitoring in distributed systems, autonomic behavior and workflow management. His Ph.D. thesis was focused on self-adaptive behavior of large-scale distributed systems based on monitoring information. He has received a Ph.D. excellency grant from Oracle and was awarded an IBM Ph.D. fellowship in 2009.

Jing Cai is an M.Sc. student at the Department of Computer Science, City University of Hong Kong. He worked with the KerData team at the INRIA Rennes–Bretagne Atlantique research center as a research intern in 2009. His research interests include distributed computing and monitoring in grid computing environments.

Gabriel Antoniu is a research scientist at the INRIA Rennes–Bretagne Atlantique research center in France. He is the leader of the KerData joint research team of the INRIA Rennes–Bretagne Atlantique research center and Ecole Normale Superieure (ENS) Cachan–Antenne de Bretagne and a member of the KerData research team. His research interests include grid and cloud distributed storage, large-scale distributed data management and sharing, data consistency models and protocols, and grid and peer-to-peer systems. Gabriel Antoniu received his bachelor of engineering degree from the National Institute of Applied Sciences of Lyon (INSA), France, in 1997, his M.Sc. degree in computer science from Ecole Normale Superieure (ENS) Lyon, France, in 1998, his Ph.D. degree in computer science in 2001 from ENS Lyon, and his habilitation for research supervision (HDR) from ENS Cachan in 2009.

Luc Bougé is a professor and the chair of the Informatics and Telecommunication Department (DIT) at Ecole Normale Superieure (ENS) de Cachan, Antenne de Bretagne, France. He is a member of the KerData joint research team of the INRIA Rennes–Bretagne Atlantique research center and ENS Cachan–Antenne de Bretagne in France. His research interests include the design and semantics of parallel programming languages and the management of data in very large distributed systems such as grids, clouds and peer-to-peer (P2P) networks.

Received: 1 July 2010
Revised: 6 December 2010
Re-revised: 21 January 2011