UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE LICENȚĂ

Replicare adaptivă a datelor în BlobSeer

Coordonatori ştiinţifici:
Prof. Dr. Ing. Valentin Cristea
Dr. Ing. Alexandru Costan

Absolvent:
Firică Alexandra - Nicoleta

BUCUREŞTI

2011

POLITECHNICA UNIVERSITY OF BUCHAREST

FACULTY OF AUTOMATIC CONTROL AND COMPUTERS

COMPUTER SCIENCE DEPARTMENT

BACHELOR THESIS

Adaptive Data Replication in BlobSeer

Thesis supervisors:
Prof. Dr. Ing. Valentin Cristea
Dr. Ing. Alexandru Costan

Firică Alexandra - Nicoleta

BUCHAREST

2011

Table of Contents

Abstract

One of the key elements in ensuring autonomic behaviour for data storage systems is adaptive data replication. This kind of behaviour is important and desirable as it can provide various advantages, such as increasing availability of highly accessed data and minimizing the storage resources used, thus leading to significant improvements in the overall performance of the system. In order to achieve autonomic behaviour, certain adjustments must be made, in accordance with the current state of the system, which is obtained by analyzing various parameters, such as load modifications, available resources, number of read/write actions per resource. This thesis addresses the problem of enabling self-adaptive data replication within the BlobSeer distributed storage system. We discuss the system's architecture, present the existing replication module and the way we intend to enhance it. We analyze the impact of the modified replication module in a distributed testing environment and we prove that it is efficient and very much responsive to the clients' needs.

Keywords: *BlobSeer, MonALISA, adaptive data replication, storage space, autonomic behaviour, replication factor, data availability.*

1. Introduction

In the course of this introduction, some general notions about the latest research and technologies will be presented, thus establishing the current trends as well as the focal points that draw the public's eye at the time of the writing of this paper.

In addition to this, the second part of the introduction is reserved for a discussion, whose purpose is to establish what the "adaptive" term means to us, as well as to the scientific community. Moreover, we explain why our solution can be considered adaptive and what benefits come from this type of behaviour.

1.1 General notions

One of the most common trends in computer systems nowadays refers to distributed computing, which includes the grid and cloud concepts. These have rapidly gained popularity in both academic and business environments, mainly because they offer data processing power as well as storage facilities. The system this paper studies, BlobSeer, is one such data storage system. As its name implies, the data is stored in the form of blobs (Binary Large OBject), which are splitted into chunks and stored on several data providers. This technique is more formally known as data striping [1] and the chunks are referred to as pages.

In the majority of cases, the processing power that this type of system provides is used mainly to analyse and transform the stored data. This is the reason why data availability is a very important issue. The apparently simple and obvious solution to this problem has been used so far, that is to manually specify a number of desired copies at the creation of the file. However, there are some disadvantages to this approach, such as not being able to modify the given number of replicas and, finally, a waste of resources, in case the replicas are never actually needed. Ultimately, in a Cloud context, this waste of resources translates into a waste of money, since the price paid for the required service and for the disks themselves varies according to how many storage resources are used.

Another important aspect to be considered is the fact that the system needs to be reliable, which means that, even though there is a possibility of a crash from a node containing information, the end user shouldn't need to suffer from it and, more importantly, data should not be lost. However, in the existing approach, if one of the copies was lost, due to any given reason (e.g.: disk failures, network failures, uneven load on the machines, etc) no action is triggered to create another copy in order to maintain the replication factor specified at the creation of the file.

This is where the adaptive replication data module steps in. Its role is not only to maintain a given number of copies of a certain object, but to also give the system the capability to decide whether there should be more or less copies in the system. Basically, what this means is that the system itself, without any outside intervention, would be able to analyse its current state and make a decision to increase or decrease the replication factor. Such a decision is taken based on the analysis of various pieces of information about the data providers, gathered using the MonALISA monitoring system and a special monitoring module developed for BlobSeer. The effect of such a capability of

decision is enormous as it translates not only into availability of data, leading to customer satisfaction, but also into reliability, thus increasing people's trust in distributed services.

1.2 Discussion on adaptivity

One of the papers used as reference for defining and understanding autonomic behavior in systems in general, as well as in distributed systems is brought forward by Kephart and Chess [2]. The aim of their paper is to stress the current situation, which is presented by them as a crisis. According to them, the main reason for this situation is the "loom-like software complexity", but also the lack of solutions for the high demand systems have (e.g.: configure, optimize, maintain, merge, etc.). Moreover, they highlight the fact that the actual trend of innovations leads to larger-sized and even more complex systems, but fails to offer a practical solution to the existing crisis.

Surely, their paper comes to offer one such solution: autonomic computing, which, in a larger sense, means "computing systems that can manage themselves given high-level objectives from administrators". A very interesting and powerful significance is held by the term "autonomic" itself, because of its biological connotation: the autonomic nervous system governs heart rate and body temperature. Furthermore, it is a know scientific fact that people who suffer from high or low blood pressure cannot control it by using their conscious brain, as this, together with the previously mentioned heart rate and body temperature, is a function of the nervous system, handled in the subconscious. That is to say that these functions are handled internally, with no need of intervention from our aware mind. We do however expect them to always happen, and, more importantly, the way these functions take place are sometimes influenced by the current string of events. The conclusion that can be drawn from these facts is that some functions can be handled by the system itself (whether it is the human body or a computer system) and done so adaptively.

At the core of the very notion of autonomic computing lies the self-management concept. It has been defined as a reunion of four aspects: self-configuration, self-optimization, self-healing and self-protection. These four concepts can be considered together, in a more general notion: self-maintenance, which also defines autonomic systems. Successfully implementing one or more of these concepts in a part of a system can lead to considering that particular part autonomic, and thus helping the entire system become autonomous, as far as the aspects addressed in the autonomic parts are concerned.

In the next paragraphs we emphasize the characteristics which lead to the conclusion that the replication module we propose can be considered autonomic. This also implies that, as far as managing replicas is autonomous in the BlobSeer distributed system.

First of all, the idea of replicating data gives the module, and, indirectly, the entire system, a self-protection capacity. This can easily be understood if one takes into consideration what the self-protection aspect means: the ability to defend against malicious attacks or cascading failures. One simple example of how replicas achieve such a goal is a hardware failure, which usually may lead to a loss of data. This is not the case when there are other replicas in the system.

Secondly, the replication module also gives the system a self-optimization capacity. As described in the paper mentioned above [2], for systems in general, self-optimization means "seeking ways to improve operation, identifying and seizing opportunities to make themselves more efficient in performance or cost". Taking this brief definition into consideration, we can easily state that our module can perform a self-optimization function for the entire BlobSeer system. We base

this statement on the fact that our module encourages not only a increase in the number of replicas, but also a decrease when some data is no longer in high demand, hence lowering the storage cost. In both cases(increase or decrease of replication factor), a performance improvement is achieved by a better response to the clients' demands – when a piece of information is frequently requested, the fact that there are copies of it in the system makes it more available to the clients, which can access it simultaneously.

As we have proved, the module brings autonomic behaviour to the system, but this important feature does not imply that the system "knows" exactly what to do without any intervention from the human administrators. As the paper itself stipulates, the desired effects must be made known to the system in the form of high-level specifications of goals and constraints. It is very important that these specifications be given accurately by the human users of the system. One example of such specifications in our case is the replication factor that the client can specify for his/her data.

1.3 Thesis objectives

The main purpose of this paper is to present the way the transition from static to dynamic replication of data can be achieved, by using an adaptive data replication module. This module is meant to be part of the project mentioned earlier, BlobSeer. This distributed service was designed to provide data storage while dealing with the requirements characteristic to large-scale data-intensive distributed applications [1], which can be resumed to two main aspects: heavy access concurrency and large unstructured data, kept as blobs.

Our aim is to enhance the already existing data replication module [3], by adding the facility of adaptively decreasing the replication factor. Thus, the adaptive replication module will be able to maintain, increase as well as decrease the number of replicas.

1.4 Thesis outline

The reminder of this thesis is organized as follows:

Chapter 2 presents a survey of the current research efforts that deal with data replication problems. It includes several algorithms and a description of two file systems that, at the present time, implement such mechanisms. A short comparison with our BlobSeer Replication Module is included, in order to highlight the various advantages and disadvantages.

Chapter 3 provides background information for the systems we used in the development of this thesis: BlobSeer, the data storage service that will include our Replication Module, and MonALISA, which we use for monitoring purposes. In this chapter we introduce basic information about BlobSeer and its components and about the way they interact in order to achieve fine grain access under heavy concurrency in a distributed storage system.

Chapter 4 presents the Replication Module's architecture and the way we have modified it, in order to implement the decrease of the replication factor. We also explain in detail the various parts that the Replication Module is made up of, such as Listeners and the Replication Manager. Our contribution in the Replication Manager is also highlighted in this chapter.

Chapter 5 elaborates on the implementation details of the Replication Module, mainly referring to the corresponding part of the Replication Manager that deals with the decrease of the replication factor. There is also an explanation on how the metadata tree is updated by the Enforcer component of the Replication Manager at the moment of a deletion of a replica, in order to ensure consistency.

Chapter 7 focuses on evaluating our component. We include the results of several tests to outline the influence that the modified Replication Module has on BlobSeer.

Chapter 8, which is the last chapter of this thesis, , presents the conclusions we've reached while developing the Replication Module, as well as some possible future improvements that can be made to our module.

2. Related work

The idea of creating replicas of frequently accessed data when dealing with read-intensive systems is not all that new, as we show below. It has been largely used in data management systems similar to BlobSeer, namely Google File System (GFS) and Hadoop Distributed File System (HDFS). The reason for this is that there are great advantages which can result from implementing replication, such as bandwidth savings, leading to a reduction in user response time, and to an overall improvement of the entire system. On the other hand, data replication implies some disadvantages when dealing with a write-intensive system, because of the extra cost due to multiple updates.

However, one must keep in mind that the algorithms and replication schemes used in various systems may differ greatly from one system to another. The replication scheme is made up by the set of providers at which an object is replicated. The reason for this great variety of algorithms, some of which are presented below, is the attempt to find an optimal replication scheme. That is to minimize the network traffic, taking into consideration the data access frequencies (reads, writes, updates, etc.), but never disregarding the particularities of the system itself (e.g.: architecture, desired functionality, etc.).

In the subchapters following, we present some file systems which already provide support for autonomic data replication system. Our aim is to focus on their key features, as well as their implied advantages and disadvantages, but especially on the way data is replicated within these systems. We particularly point out the way in which the decrease of the replication factor is made.

2.1 SRA, AGRA

In the paper written by Thanasis Loukopoulos and Ishfaq Ahmad [4] several algorithms which deal with the Data Replication Problem, DRP, are described. The first idea to deal with this problem was a static algorithm, called SRA. It is an algorithm based on a greedy method and is considered to be particularly efficient within systems with high read concurrency. During this algorithm, which is applied in stages, a benefit value is computed per storage unit. This value helps the system decide whether an object is to be replicated at a certain site in the network or not. This means that an object can be replicated at a site only if two conditions are met: the remaining storage capacity of the site must be greater than the size of the object and the benefit value computed must be positive.

However, the static replication algorithm does not behave desirably when a large number of writes takes place in the system. This is the reason AGRA was born. AGRA stands for adaptive genetic replication algorithm and is a dynamic algorithm inspired by Genetics, in the sense that the initial replication scheme is associated with the initial population of chromosomes. AGRA can be regarded as a search method based on the evolutionary concept of natural mutation and survival of the fittest. Three different genetic search operations are applied in order to transform an initial population, with the objective to improve its quality: selection, crossover and mutation. The notion of chromosome represents an encoded representation of a feasible solution, most commonly structured as a bit string. AGRA adapts to the changing environment very quickly

and readjusts the replication scheme with solutions that are comparable to static algorithms, according to [4].

As far as the decrease of the replication factor is concerned, the main method used is to choose randomly which objects should be deallocated. The authors have also suggested using a greedy method and calculating the negative impact each possible one object deallocation might have. However, in order to estimate how beneficial a replica is, both global properties of the object and local characteristics must be taken into consideration. The global properties consist of whether the object is read demanding or not and how many of its replicas exist in the network. Naturally, an object having high update ratio, but being widely distributed will have more chances of being selected for deallocation. Also, large objects with poor local read demand are preferred for deallocation, since this will result in freeing up more space for future allocation.

The solution we propose for decreasing the replication factor is based not on estimates, as AGRA, but on real time measurements of the various parameters involved, such as reads, writes, load, etc. Therefore, we can state that our method makes a decision in a more informed manner. Moreover, the way the Replication Module is implemented, it can automatically adjust to Data Provider failures and maintain the replication scheme.

2.2 Google File System

The Google File System (GFS) [5] is a cluster-based distributed file system whose main goal is to satisfy the needs of the MapReduce paradigm. A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. The data files are huge and, therefore, divided into fixed-size chunks and distributed among chunkservers. The chunk size is 64 Megabytes. Compared with the role of the chunkservers, the role of the master is more complex, as it implies assigning unique chunk handlers each time a new chunk is created and also storing metadata and interacting with the chunkservers through an exchange of messages, according to the HeartBeat. As a result of the communication between the master and the chunkservers, status information is collected and used in making various decisions, such as replication or chunk placement decisions.

An important difference between the GFS and BlobSeer is that, while the GFS uses a fixed-size chunk, in BlobSeer the chunk size is specified at the time of blob creation. More importantly, its size is usually a multiple of the data size the client assumes to process in one step. This is an obvious BlobSeer advantage, as the GFS approach, although it implies less physically stored metadata, it also favours internal fragmentation.

Another important aspect in which the two systems differ is the way the system is able to react to high concurrency, which involves both reading and writing operations. Both systems choose to create new data rather than to overwrite the already existing one. However, although the GFS allows multiple clients to append data to the same file concurrently while guaranteeing atomicity, concurrent writes to the exact same region are not serializable, thus leading to a possibly mixed and unreliable content of the updated region. This is not the case with BlobSeer, as it efficiently handles concurrent writes at arbitrary offsets in the same blob, as described in Chapter 3, Section1.

As far as the dynamic replication problem is concerned, the GFS stores three replicas for each file, but users can specify other replication levels. All chunks of files are re-replicated by the master, as soon as the number of available replicas drops under the user-specified goal, in an attempt to maintain the desired replication factor. There can be many reasons for the sudden

unavailability of a replica: a chunkserver may be unavailable, the replica itself may be corrupted or a hard disk may be disabled. When re-replicating chunks, a priority is assigned to every one of them, based on several factors. One of the factors is how far the chunks is from its replication goal. For example, a chunk that has lost one replica is assigned a lower priority than one that has lost two replicas. Moreover, the system automatically increases the priority of any chunk that is blocking client progress, in order to minimize the impact of system failures on running applications.

An additional aspect when dealing with replicas is their placement in the system. Some criteria are taken into consideration, such as equalizing disk space usage and spreading replicas across racks. The motivation for implementing this kind of replica placement policies is achieving the maximum data reliability, as well as the best network bandwidth possible. Replicas are also rebalanced by the master, who examines the current replica distribution and moves replicas for better disk space and load balancing.

The master also checks whether the replicas are up-to-date. The replicas that are found to no longer be up-to-date are considered “stale” and are marked for deletion. The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information [5]. In addition, it is also the master’s prerogative to choose which existing replica to remove. Generally, it prefers to remove those replicas found on chunkservers with below-average free space, in order to equalize disk space usage.

2.2 Hadoop Distributed File System

The Hadoop File System [6] is the underlying storage layer of Hadoop open-source MapReduce implementation. Its purpose is to store very large data in a reliable manner across machines in a large cluster and to achieve transfers of those data at high bandwidth. Each file is stored as a sequence of blocks. All blocks in a file, except the last block, have the same size. It is a cluster-based distributed file system, and, therefore, data and metadata are decoupled. While metadata is stored on a dedicated server, called the NameNode, data is stored on DataNodes. Besides maintaining metadata, the NameNode also has the role of monitoring of each DataNode, meaning to keep a record of the total storage capacity, the fraction of storage used and the number of data transfers currently in progress. All this information is used to enable efficient space allocation and load balancing strategies. But, most importantly, the NameNode must serve client requests. There are some know disadvantages to this approach, which include scalability and data availability issues. The explanation for these situations is that the NameNode becomes unresponsive as the number of files and concurrent requests grows. Furthermore, the NameNode can be regarded as a single point of failure for the system.

On the other hand, BlobSeer does not have this type of issues, as the role that the NameNode has is divided in BlobSeer between the Provider Manager, whose main task is to keep track of the Data Providers registered in the system, and the Metadata Providers, who store information about the existing blobs in a distributed fashion.

Another major difference between HDFS and BlobSeer is that HDFS implements a single writer, multiple readers model. BlobSeer, on the other hand, efficiently supports concurrent writes to the same object because of its versioning based concurrency control.

As far as the dynamic replication problem is concerned, in the HDFS the replication factor is configurable per file. The number of replicas for a file can be specified at the moment the file is created, but can also be changed at a later time. All decisions concerning the replication of blocks (pieces of files) are taken by the NameNode. HDFS's reliability and performance relies mainly on the placement of replicas. The most common case of replication factor is three. That is to say that the system must decide where to place the three replicas. In this case, the HDFS has a specific policy: one replica is placed on one node in the local rack, another on a different node in the local rack, and the third one in a different node in a different rack. This policy cuts the inter-rack write traffic which generally improves write performance, without compromising data reliability or read performance.

When trying to satisfy a read request, HDFS tries to access the replica that is closest to the reader. Thus the global bandwidth consumption and read latency are minimized. In case a replica exists on the same rack as the reader node, that replica is preferred.

As it can easily be noticed, the two systems are alike, in the respect that both allow the replication factor for a blob/file to be specified at creation. However, BlobSeer's approach is based on the read/write fluctuations, being able to dynamically increase or decrease the replication factor. This means much more efficient storing, by using the storage space available for frequently requested data and not keeping it occupied with copies of unrequested files. Indeed, BlobSeer's Replication Module can make a well-informed decision when modifying the replication factor, as it has a up-to-date view of the system, thus bringing a series of advantages to this distributed file system: reliable data storage, easily adaptable to failures a dynamically-adjusting replication scheme that is optimized to use less storage resources.

3. Background

3.1 BlobSeer

BlobSeer [7][8] is a highly scalable data storage and management service designed to deal with requirements of large-scale data-intensive distributed applications. In this section we describe the system's architecture, the way interactions take place between its entities and the access interface put at the disposal of the clients. Moreover, we stress the main features that make BlobSeer different from other distributed systems and contribute to the enhancement of concurrency.

3.1.1 BlobSeer's Architecture

Data in BlobSeer is abstracted as a large sequence of bytes and stored as blobs (Binary Large OBjects), meaning very large files. Similarly to other data storage services, BlobSeer addresses the well-known storage challenges, which include heavy access concurrency, data versioning, large aggregated storage space and fine grain access.

What truly makes BlobSeer stand out from all other data storage services is that it is the only one that supports concurrent writes, reads and appends, while versioning the files and providing fine grained access to them.

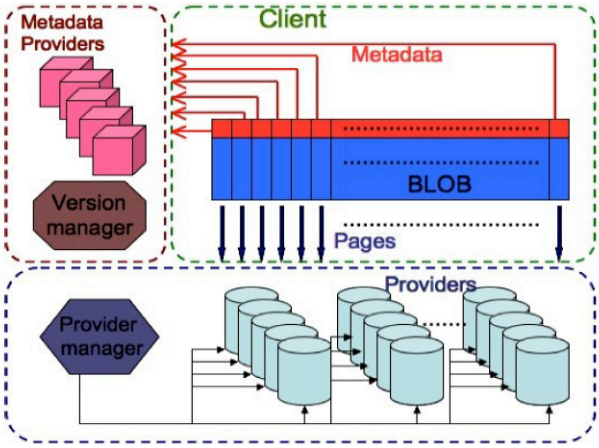


Figure 1. BlobSeer's Architecture

BlobSeer's infrastructure[7] consists of a set of distributed processes, that interact with each other in various ways, as described in the next section. The various roles "played" in the system are described below.

Clients may perform various operations, such as creating, reading, writing and appending blobs. There may be multiple concurrent clients, both for reading and writing new data, and their number may dynamically vary in time.

Data Providers physically store the pages generated by the write and append operations. New data providers may dynamically join and leave the system.

The Provider Manager keeps information about the available storage space and schedules the placement of newly generated pages according to a load balancing strategy. The main purpose behind this type of strategy is to ensure a balanced distribution of data in the system, thus eliminating the possibility of overloading one Data Provider.

Metadata providers physically store the metadata allowing clients to find the pages corresponding to the blob snapshot version. Metadata is organized as a segment-tree like structure and is scattered across the system using a Distributed Hash Table (DHT). The distribution of data and metadata is another key characteristic that differentiates BlobSeer from other data storage services. More detailed information about metadata can be found in Chapter 5.

The Version Manager is the key actor of the system. It registers update requests (append and write operations), assigns snapshot version numbers, and, eventually, publishes new blob versions, while guaranteeing total ordering and atomicity.

3.1.2 Interactions between BlobSeer entities

The interactions between BlobSeer's entities are illustrated below, in Figure 2. They can easily be explained by observing and understanding how the read and write operations work[7][9].

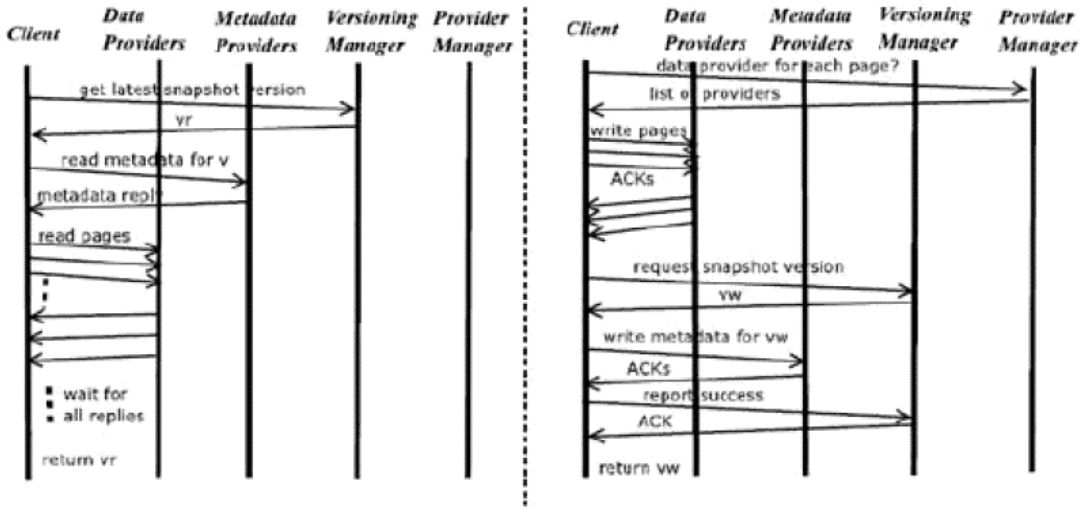


Figure 2. Interactions inside BlobSeer:
READ (left) and WRITE (right)

Reading data In order to read data, clients must first contact the Version Manager. Some information is required from the client, such as a blob id, a specific version of that blob, and a range, which translates into an offset and a size. If the version has been published and is, therefore, available, the client then queries the Metadata Providers for the metadata indicating on which providers the corresponding pages are stored. Finally, the client fetches the pages in parallel from the respective data providers.

Writing and appending data In order to write data, the client first determines the number of pages that cover the range to be written. It then contacts the Provider Manager and requests a list of Data Providers able to store the pages. Afterwards,in parallel, the client generates a globally unique page id for each page, contacts the corresponding page provider and stores the contents of the page on it. After an acknowledgment is received from each Data Provider, the client contacts the Version Manager and requests a new version number. This version number is then used by the client to generate the corresponding new metadata. Finally, the client notifies the Version Manager of success, and returns successfully to the user. At this point, the Version Manager is responsible for eventually publishing the new version of the BLOB.

One may consider the append operation as a particular case of write, where the offset is implicitly the size of the previously published snapshot version. The detailed algorithms for the described operations are given in [7].

3.1.3 Access interface

As mentioned above, clienta are allowed to create blobs, to read from already existing blobs starting from an offset and to write or append a range to the blob. All these operations can be performed by the client simply by using the given interface. In this section, we describe the primitives available, including the newest addition to the system, a delete primitive.

To create a blob, clients must call the CREATE primitive.Because blobs are uniquely identified in the system, this primitive returns the id associated to the newly-generated blob.

id = CREATE()

In order to read data, the client must provide certain parameters: the blob id and version, the desired range, which translated into an offset and a size, and a local buffer where the fetched data will be placed.

READ(id, v, buffer, offset, size)

Things happen in a similar fashion in the case of the WRITE and APPEND primitives. The only difference is that the buffer provided is now used as the source for the written data. As the APPEND primitive is regarded as a particular case of WRITE, the offset is no longer needed. It is implicitly considered to be the size of the last previous version.

vw = WRITE(id, buffer, offset, size)

va = APPEND(id, buffer, size)

Some helping primitives are:

v = GET_RECENT(id)

size = GET_SIZE(id, v)

whose rolea are to provide the latest version of the desired blob, respectively to return the exact size of the blob snapshot corresponding to version v of the blob identified by id.

The most recently added primitives are those which allow the client to delete data from BlobSeer[10]. Both primitives are described below.

To delete a version of a particular blob, a client must call the DELETE primitive:

DELETE(id, v)

To delete old data, meaning all snapshot versions lower than a particular version labeled v, a client must call the primitive:

DELETE_EARLIER(id, v)

The simplicity of access interface comes to show once more that the user does not have to be aware of data or metadata location as these are handled transparently by the storage service. More than that, versioning is provided at the application level so that one client can read from any published snapshot of a blob while other generates new versions of the same blob without the need for synchronization. The corresponding algorithms of this series of primitives are presented in detail in [11].

3.2 MonALISA

In order to make an informed decision to decrease the replication factor of a file at version level, we use data collected by MonALISA[12].

MonALISA, which stands for Monitoring Agents in a Large Integrated Services Architecture, is a framework based on Dynamic Distributed Service Architecture. Its main purpose is to provide complete monitoring, control and global optimization services for complex systems.

The MonALISA system is designed as an ensemble of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services, and are able to collaborate and cooperate in performing a wide range of information gathering and processing tasks. Its features and flexibility recommend it for monitoring in large distributed systems, as shown here [13].

In our project, MonALISA plays a very important role in both directly providing information about the load and the free space of each data provider and computing metrics based on data collected from the agents that monitor the providers, thus determining the changes that take place in the read/write frequencies for a file.

All the metrics data obtained will be combined and used by the Adaptive Data Replication Module, which is described in Chapter 4 below, in order to accurately decide whether to increase or decrease the replication factor of a blob. To add MonALISA support, we used the MonALISA Service and the MonALISA Repository. The MonALISA Service receives information regarding the number of reads and writes for a blob through BlobSeer's monitoring support [14] . The MonALISA Repository is used to monitor the data providers for parameters such as free space and load. More detailed information can be found in Chapter 5, which describes the implementation of our proposed module.

4. Adaptive Replication Module

The main purpose of this module is to coordinate the creation and/or deletion of replicas in the system, thus making it more responsive to client demands and more stable as far as the ability to react to faults is concerned. As this whole module is relatively new, compared with the rest of the system, its components as well as the interactions between them and the already existing parts of BlobSeer are described below.

4.1 Replication Architecture Overview

The Adaptive Data Replication Module architecture [3] is illustrated in Figure 3. The Replication Module brings two new elements to BlobSeer: Listeners and the Replication Manager. These two new entities can be considered distributed communicating processes, in the exact same way Data Providers, Metadata Providers, the Provider Manager and the Version Manager are regarded.

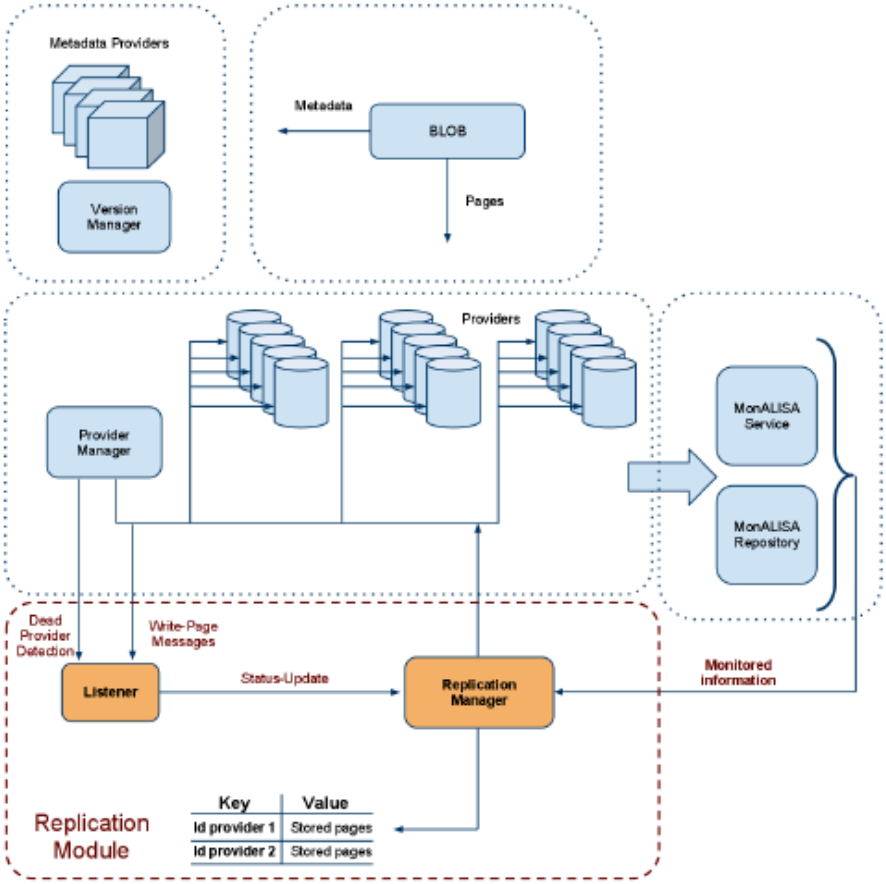


Figure 3. Replication Module

Listeners The role of the Listeners is to send status update messages to the Replication Manager in order to ensure that it has all the necessary information it requires to function properly. These Listeners can be regarded as intermediaries between the Replication Manager on one hand and the Provider Manager and the Data Providers on the other.

Status-update messages sent from the data providers contain information about the availability of a provider and the write-page events which are triggered when content is being written to a page.

The status-update messages sent from the Provider Manager are triggered immediately after the Provider Manager acknowledges the unavailability of a storage provider.

The Replication Manager This is the main component that, combined with the Listeners described above and helped by the monitoring contribution of MonALISA Service and Repository, decides whether and how replication has to take place in the system. Its role is to maintain, increase and decrease the replication factor for the existing blobs, based on the information it receives from the Listeners, which it analyses and compares with the results computed internally, by parsing logs and determining whether the number of page-reads or page-writes indicates a need for a increase/decrease in the replication factor .

4.2 Illustration of the functionalities of the Replication Module

The Replication Module is currently able to maintain and increase the replication factor[3]. In the next two subchapters we present the capabilities of the Replication Module before our alterations were performed. This capabilities refer mainly to the ability to maintain and/or increase the replication factor of blobs. Our contribution, first schematically, and afterwards in more detail, is presented beginning with subchapter 3 – Zoom in on the Replication Manager. Our aim is to focus entirely on instrumenting the decrease decision for the blobs in the system, by taking into consideration both the monitored data and the client's needs.

Below we show some scenarios when maintenance or increase are considered to be needed by the Replication Manager, with the help of the Listeners and by analyzing the data received from the monitoring service, namely Monalisa, using the filters specifically designed to monitor BlobSeer-oriented data (e.g.: reads/ writes for a blob or a page, load, etc.).

4.2.1 Maintaining the replication factor

The first important aspect that the Replication Module was meant to solve was the ability to maintain the replication factor. This was achieved by designing the Replication Manager to communicat with the Provider Manager, the the Metadata Providers, as well as with the Data Providers themselves. Every available Provider announces its presence to the Replication Manager. Moreover, every time a write occurs on a Provider, the Replication Manager is directly informed and stores this information for every Provider. Thus, when the Provider Manager indicates that a Data Provider is no longer available(for hardware or software issues, or simply because it has left the system), the Replication Manager knows exactly what pages of what blobs were stored on the said

Provider and is able to replicate elsewhere, provided that the only existant copy was not on the unavailable Data Provider.

What this achieves is a much needed fault tolerant solution implemented in a system in which the described types of failure are more likely to be the rule than the exception. Below we present a typical scenario in which the decision to maintain the replication factor is made.

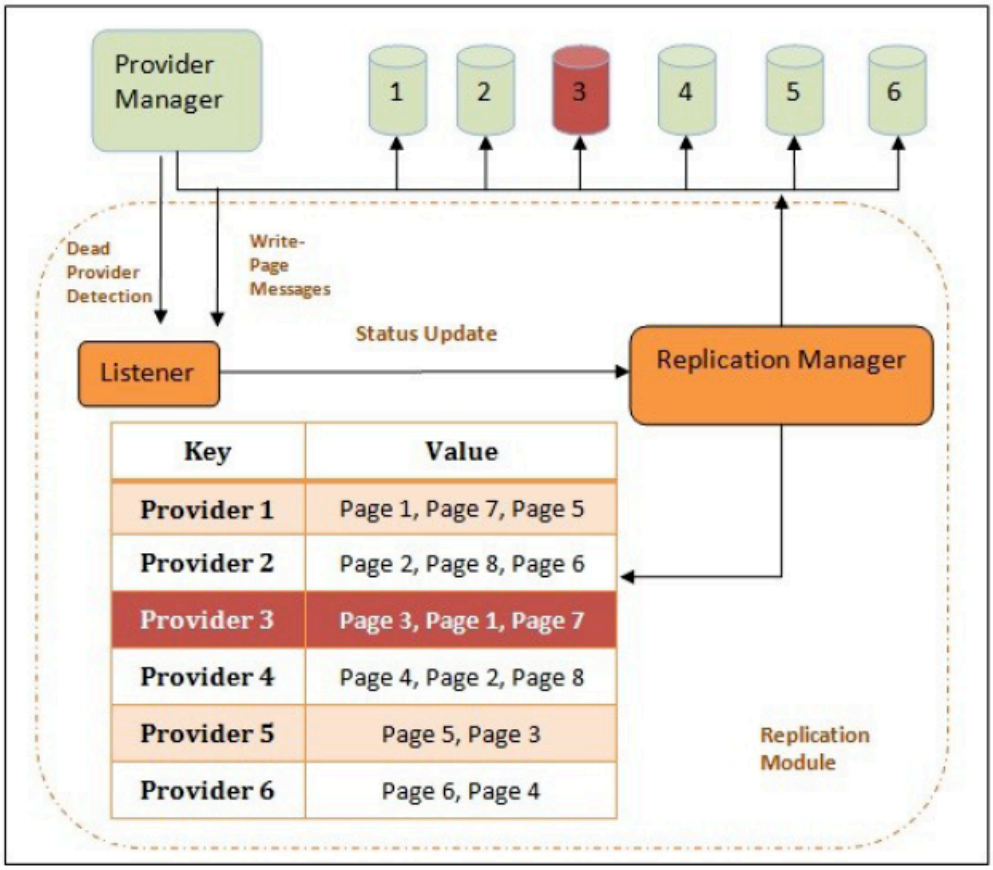


Figure 4. Maintaining the replication factor

Here, we can see that there are 6 Data Providers available in the system at one time. There is only one blob and it has a desired replication factor, given on creation, of 2. At a later point in time, the third Data Provider fails and this is where the Replication Manager steps in. It takes action to create copies of the first, third and seventh page on the other Data Providers, thus ensuring that the replication factor for all the pages that make up the blob is maintained. However, one most take into consideration that there are no duplicates accepted for a page. In the case presented above, this means that Page 1, which needs to be copied somewhere else, will be placed anywhere else except on Provider 1, because it already contains a copy of Page 1.

Not accepting duplicates on a Data Provider means not allowing a single point of failure and, consequently, not losing all the available replicas of a page in the case of a crash. On the other hand, this can become an issue when the number of providers in the system is low and there are no

providers left to receive the pages. This issues has been dealt with by adding a vector that contains each key that the Replication Manager was unable to store a copy of. It is the first to be analysed when a Data Provider becomes available.

4.2.1 Increasing the replication factor

The second important aspect the Replication Manager deals with is deciding when to increase the replication factor of a blob. This decision is made by analyzing various monitored parameters, such as load5, bandwidth and free memory, for every Data Provider in the system. The replication factor is increased for a blob when the Data Providers that store its pages are under very much pressure, meaning that there is not much free memory left, bandwidth is high, etc. The explanation for this method is that a Data Provider found in the conditions mentioned above is more likely to fail than the others.

In the picture below, we have such a scenario, which illustrates the instrumentation of the decrease decision based on the load of the machines.

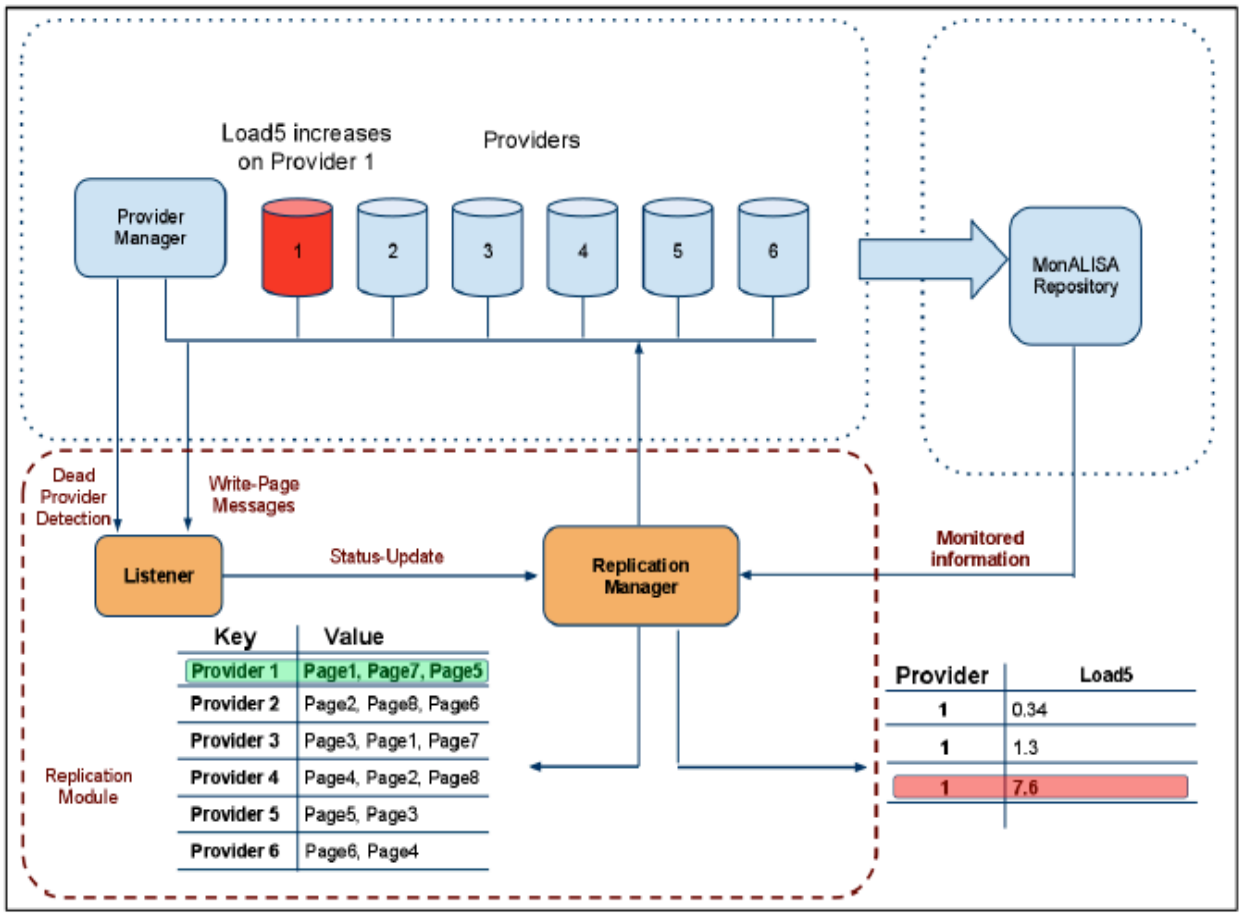


Figure 5. Increasing the replication factor

This example shows a blob with 8 pages, with a replication factor of 2. The monitoring data is taken from the MonALISA Repository through a PostgreSQL database. Because the load has increased on provider 1, the pages kept on it will also be copied elsewhere, increasing thus their replication factor. The Replication Manager reaches this decision by taking into consideration the value 4, which has been established as the normal threshold for load5.

The Replication Manager may also take into consideration other parameters, such as the number of reads registered for a blob. It is reasonable to increase the replication factor of a blob by looking at the overall demand for its contents and establishing in this manner the general interest for that content.

4.3 Zoom in on the Replication Manager

The Replication Manager also cotains four new components, which we describe in the following paragraphs. The main purpose behind introducing these new components was addressing the issue of decreasing the replication factor, depending on the state that the system is in. That is to say making the decision of marking as deleted a replica in the metadata tree and enforcing this decision by actually deleting the information from the selected Data Provider. It should be mentioned that it is very likely that more than one copy be submitted for deletion at a given time.

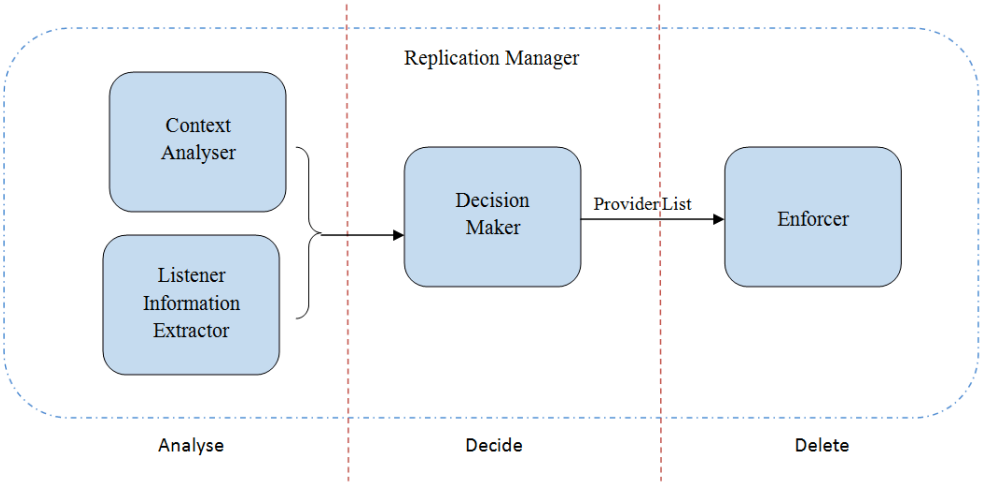


Figure 6. Replication Manager – decrease replication factor

ContextAnalyser Its main purpose is to keep track of the given situation scenarios and interrogate the MonALISA database in order to discover whether any of the predefined scenarios is applicable in the current situation. In the first stage of the implementation of this project, the scenarios are kept as simple text files. Each of them specifies a certain rule. For instance, by analysing the monitoring data, the system can decide whether the number of reads/writes per time unit for a page is under a certain limit, specified in a scenario. If the system comes to the conclusion that one or more scenarios are present, it returns a list of data providers from which to remove the page in question, in order to decrease the replication factor.

However, one should keep in mind that the time unit may vary, according to how reactive the system is desired to be, or by taking into consideration the type and frequency of the operations that take place. For example, the time unit may be set anywhere between ten minutes and a given number of days.

ListenerInformationExtractor Its aim is to collect information from the Listeners and interpret it. An example of its use is the monitoring of the write operations performed. If the ListenerInformationExtractor finds the number of writes below a certain limit for a particular page, it can suggest one or more data providers to have their replicas of that page removed. Another case in which this component might decide to decrease the replication factor is when it receives information regarding the low free space of one or more data providers.

DecisionMaker It compares the results given to it by the two previous modules (the ContextAnalyser and the ListenerInformationExtractor) and thus reaches a decision whether a decrease in the number of replicas is indeed required. If the decision is to go forward with the decrease of the replicas, a list of data providers is passed on to the Enforcer.

Enforcer It implements the deletion itself from the Data Provider(s), as well as the modification of the metadata tree, as we describe in Chapter 5, together with other implementation details. Although the decision to decrease the number of existing replicas is taken at a version level, it may include only some of the pages characteristic to a version, that is to say not all of the pages will have the same replication number, but different ones according to their usage. This is one of the most obvious aspects that suggest the adaptive nature of the system as enforced by the Replication Manager.

4.4 Illustration of the solution

In the example illustrated below, we show a typical situation in which the Replication Manager, or, more specifically, the Decision Maker component, would decide to decrease the replication factor for a blob.

The Data Providers are monitored using the MonALISA Service. The obtained data is stored in the MonALISA Repository and it can easily be accessed through sql queries made to the PostgreSQL database server used by the Repository. The parameters used may vary. We have used load5, bandwidth and free space. In the example given, however, we highlight only those monitored parameters that are in accordance with those parameters mentioned in the scenario. Detailed explanations about the possible scenarios can be found in subchapter 4 of the current chapter.

The Replication Manager communicates with the Provider Manager, which informs it about the Data Providers that are no longer available, using the Listener as an intermediary. It also receives information directly from the Providers when they become available in the system and when a write event takes place. This explains how the Replication Manager knows the exact association (key - value) between the Providers and the exact pages that are stored on them.

One of our main goals was to make the system as responsive as the client wants it to be. We enabled this type of behaviour by giving the Replication Manager direct access to the clients' requirements. The clients can express the behaviour they expect the system to have by specifying various parameters' limits, structured as scenario files. In our project, the scenarios are kept as simple text files, their content being interpreted as rules by the system. Basically, this means that, as long as those rules are obeyed, the system can take whatever action it decides. However, when

those rules are broken, certain actions must take place. One must keep in mind that, although there are more parameters in a scenario file, and, ergo, more rules, it is a must that all rules be broken before action can to take place. For instance it makes no sense to decrease the replication factor for a blob, when, even if there are no reads, there are write actions accessing it. This indicates the need for that particular blob is still high.

In the scenario below, 5 parameters are taken into consideration: replication factor, number of reads and writes, load and query interval. For instance, the number of minimum replicas the client wants to have in the system at all times is 1. This parameter can be considered redundant, as there is always at least one copy available. However, taking into consideration the most recent work concerning the delete primitives, one might interpret the given replication factor 1 as an indicator of never deleting that data. Also, the client mentions there must be at least one read per interval, but no write is required.

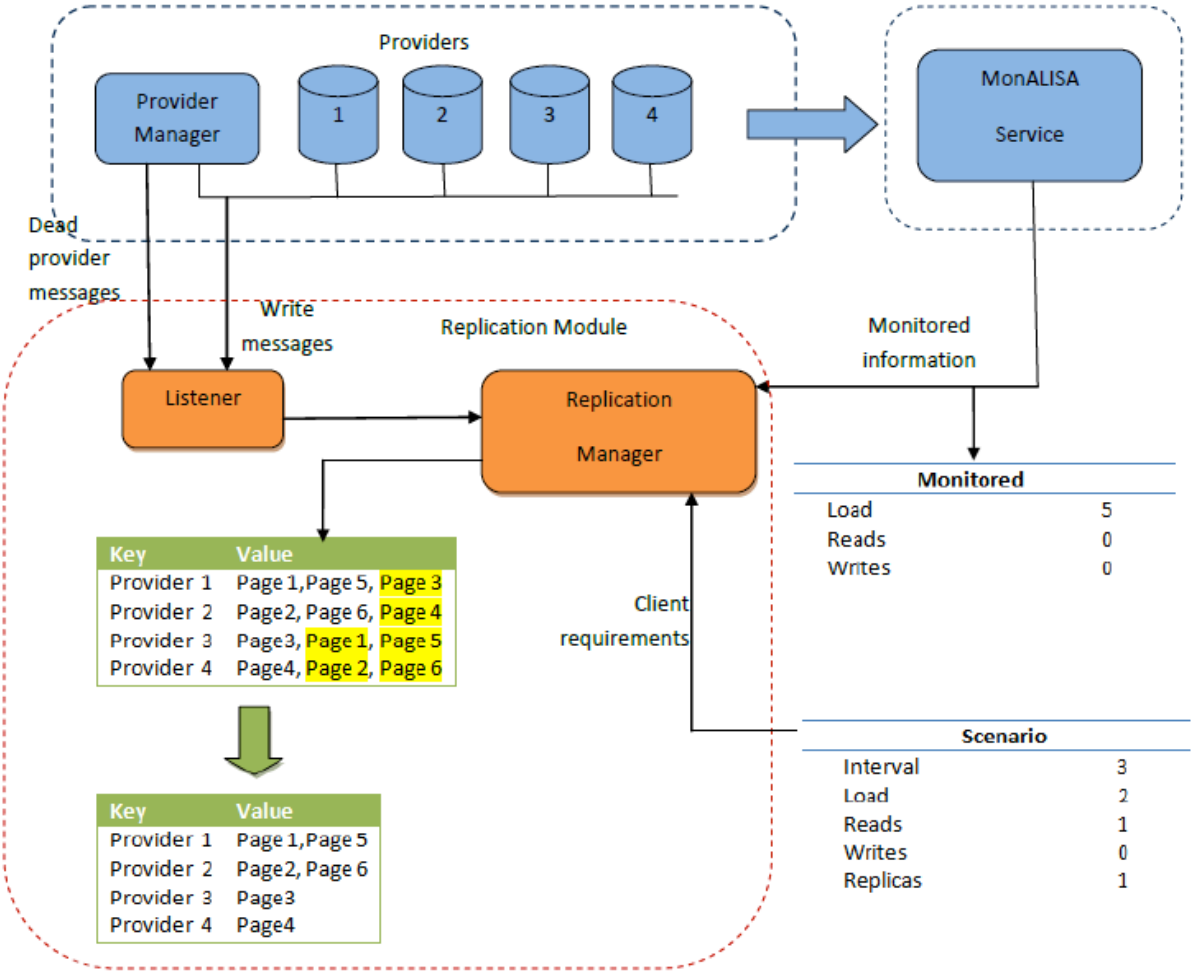


Figure 7. Replication Manager – scenario to decrease replication factor

In our example, there are 4 Providers and one blob made up of 6 pages. The current replication factor of the blob is 2. The chosen number of Data Providers is only to enhance the simplicity of the example. Our solution scales very well no matter how large the available numer of Providers is. Although the scenario file mentions 1 necessary replica, this only indicates the minimum required number of replicas. We might imagine that the blob's replication factor has been previously decided by the Replication Module. By comparing the monitored information and the information extracted by the Listener with the client's scenario, the DecisionMaker component of the Replication Manager triggers the decrease replication factor action, as it can be seen above, in the modified page table.

Another important aspect we need to highlight is that the interval parameter is the one that actually controls the flexibility of the system, as it indicates the frequency with which the database is interrogated. The longer the time period is, the less reactive to changes is the entire system.

4.5 Modifying the metadata tree

Metadata[7] stores information about the pages which make up a given blob, for each generated snapshot version. In order to keep the versions as transparent as possible, BlobSeer creates new metadata every time an update takes place. This helps provide support for heavy concurrency, as it favors independent concurrent accesses to metadata without synchronization.

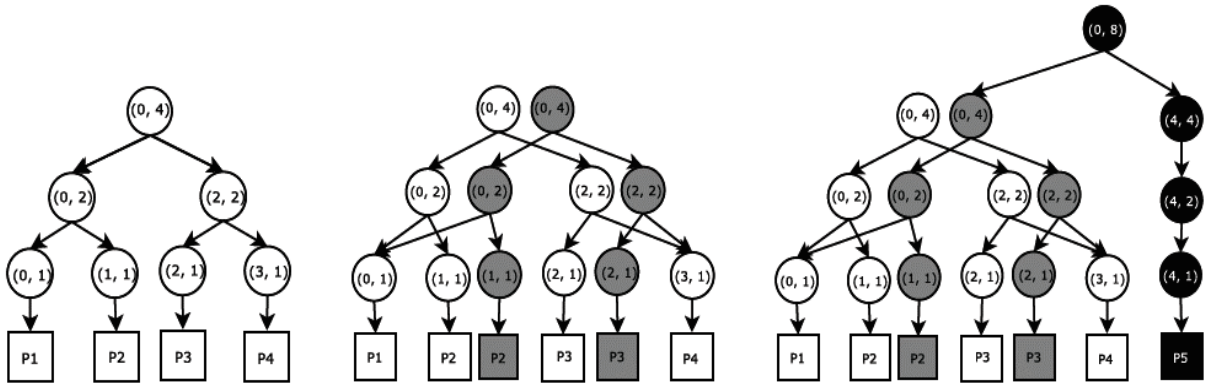
Metadata is organized as a distributed segment tree. There is one such tree associated to every snapshot version of a given blob id. A segment tree is a binary tree in which each node is associated to a range of the blob, delimited by offset and size. A node is said to cover a range, defined by a (offset, size) pair. Each leaf of the tree is considered to cover one single page.

Figure 5(a) below shows the structure of the metadata for a blob consisting of four pages. The root of the tree covers the range (0, 4), while each leaf covers exactly one page in this range.

The metadata tree is created when the first pages of the blob are written, for the range covered by those pages. In order to avoid overhead, new tree nodes are created only for the ranges that do intersect with the range of the update.

Figure 5(b) below shows how metadata evolves when pages 2 and 3 are modified for the first time.

The append operations make the blob "grow", as shown in Figure 5(c). Here, new metadata tree nodes are generated, to take into account the creation of a fifth page by an append operation.



(a) The metadata after a write of four pages

(b) The metadata after overwriting two pages

(c) The metadata after an append of one page

Figure 8. BlobSeer's Metadata Representation

Taking into consideration the way metadata is stored, our implementation of the adaptive Replication Module involves also updating the metadata tree. This step is taken by the Enforcer component from the Replication Manager. In this case, updating means traversing the metadata tree until the leaves are reached, look for the given providers in the list of providers that is found in every page and either delete them entirely from the list or just mark them as not available anymore, for that particular page. This entire method is presented in Figure 6 below.

Afterwards, there is one last important step : informing the root of the metadata tree of the modifications made, because it needs to modify the replication_count accordingly, thus maintaining consistency.

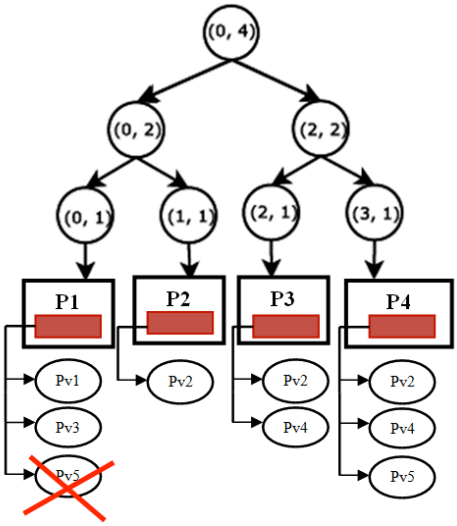


Figure 9. Modifying the metadata tree

4.6 Algorithm used

Below we show in pseudo-code the algorithm used by the Replication Manager when deciding whether to decrease the replication factor or not.

Algorithm 1 DECREASE REPLICATION FACTOR

```
Require: The monitoring database
Require: The scenario file
 1: read scenario file and interpret it

 2: repeat from interval to interval
 3:         query database -> result
 4:         for each element in result
 5:                 compare monitored data with scenario information
 6:                 if all rules are broken
 7:                         decide to decrease
 8:                         for all pages in blob
 9:                                 call Enforcer component to decrease replication factor
10:                         end for
11:                 endif
12:         end for
13: end repeat
```

The first thing the Replication Manager does is to read the scenario file provided by the client. The scenario files are in accordance with what is described in the next chapter. For the current implementation, the Replication Manager reads and interprets only one such file, and assumes the limits given there are applicable for all the blobs in the system. As a future improvement, we believe it is possible to alter the create primitive and provide it with an additional parameter – the scenario file for the blob.

The next step is to query the database for updates from interval to interval. The frequency at which the database is interrogated is given by the client in the scenario file. This is a very important aspect, which ensures the system is as reactive as the client wants it to be. For every new element in the query's result, the Replication Manager compares the monitored information with the one provided in the scenarios. This part of the application corresponds to the DecisionMaker described earlier. The decision to decrease the replication factor is made by taking into consideration all the rules the client chose to include in the scenario files. This method ensures all decisions are taken when they are really needed and the unnecessary deletion of a copy, only to increase the replication factor in the next step of the algorithm, is prevented.

The final step is to call the Enforcer component to physically decrease the replication factor. The way this component works is explained in detail below, after a pseudo-code for its' steps is presented.

Algorithm 2 ENFORCER DELETE

```
Require: The page
Require: The provider/list of providers
1:  if page is in the unresolved key list then
2:          decrement unresolved replicas
3:          if remaining unresolved replicas == 0 then
4:                  remove page from unresolved key list
5:          end if
6: else
7:          delete (invalidate) page on the given provider(s)
8:          update metadata
9: end if
```

When the Enforcer has to delete one replica, it first looks in the unresolved key list to see if the respective page is already there. If this is the case, this means that, at a certain moment in the past, the Replication Manager decided to increase that page's replication factor. However, for various reasons, this was not possible at that particular time and, therefore, the page was added to the unresolved key list. It makes perfect sense, when decreasing the replication factor by 1 to first decrement the unresolved replicas, because otherwise, things would happen as follows: a replica would be deleted, and afterwards, the Replication Module would try to see whether there are any unresolved pages which can be now copied on the existing Data Providers, taking advantage of the now free space that exists and respecting the rule to not have duplicates of the same page on one Data Provider. Basically, the decrement and, eventually, delete of a member of the unresolved key list can be regarded as a "soft" deletion of replicas, bringing the advantage of fewer operations and messages exchanged in the system, thus reducing the overhead.

If the page whose replication factor has to be decreased is not in the unresolved key list, a Data Provider that contains a replica of the page is looked up and the replica is invalidated. Afterwards, the metadata tree is updated, as was explained in detail in the previous subchapter.

5. Implementation details

The Adaptive Data Replication Module [3] was implemented as a separate module that communicates with the Provider Manager, the Data and Metadata Providers. What we have done is to modify the already existing Replication Module in order for it to be to provide efficient decrease of the replication factor. We have done so by adding several components, such as the Enforcer and the Decision Maker, which were described in detail in the previous chapter.

Our implementation is for C++, making it thus easier to integrate with the rest of the BlobSeer modules.

5.1 Data structures and interactions scheme

This section introduces the data structures involved in our implementation with the help of UML class and sequence diagrams. Some attributes and methods have been left out, and only those relevant to our implementation have been shown in the diagram below. The second diagram gives further details on the interactions which take place between the Replication Module and the other entities of the system during one interval (meaning every time the watchdog queries the data base and finds updates).

The class diagram presented in Figure 10 below shows the most important part of the composition of the Replication Manager class. Its aim is to do this as accurately as possible, although some minor fields characteristic to the class have been left out of the diagram for simplicity reasons. Moreover, it comes to show the exact relationship between the fields and methods of the Replication Manager class and the ones that already exist in the system.

For instance, the type query_t is used in the definition of the unresolved keys vector. The query_t class is part of the common data structures. Its purpose is to help define a read or a write request, while also being able to completely identify a data page or a metadata node. This is the reason why this class can be regarded from two different angles, which also can be seen in the class diagram. The fields that this class contains basically speak for themselves, as far as their use is concerned: offset, size, version and id are obviously fields that are used to define a page in BlobSeer, but also the location of the distributed metadata.

The first point of view is the one of the root_t class, which defines its node field as a query_t object. On the other hand, the dhtnode_t class uses the query_t class to define a node containing metadata information. Our implementation uses both points of view. The first one is used when analyzing read and/or write messages sent across the system. The second perspective is used mainly when altering the metadata tree as shown above.

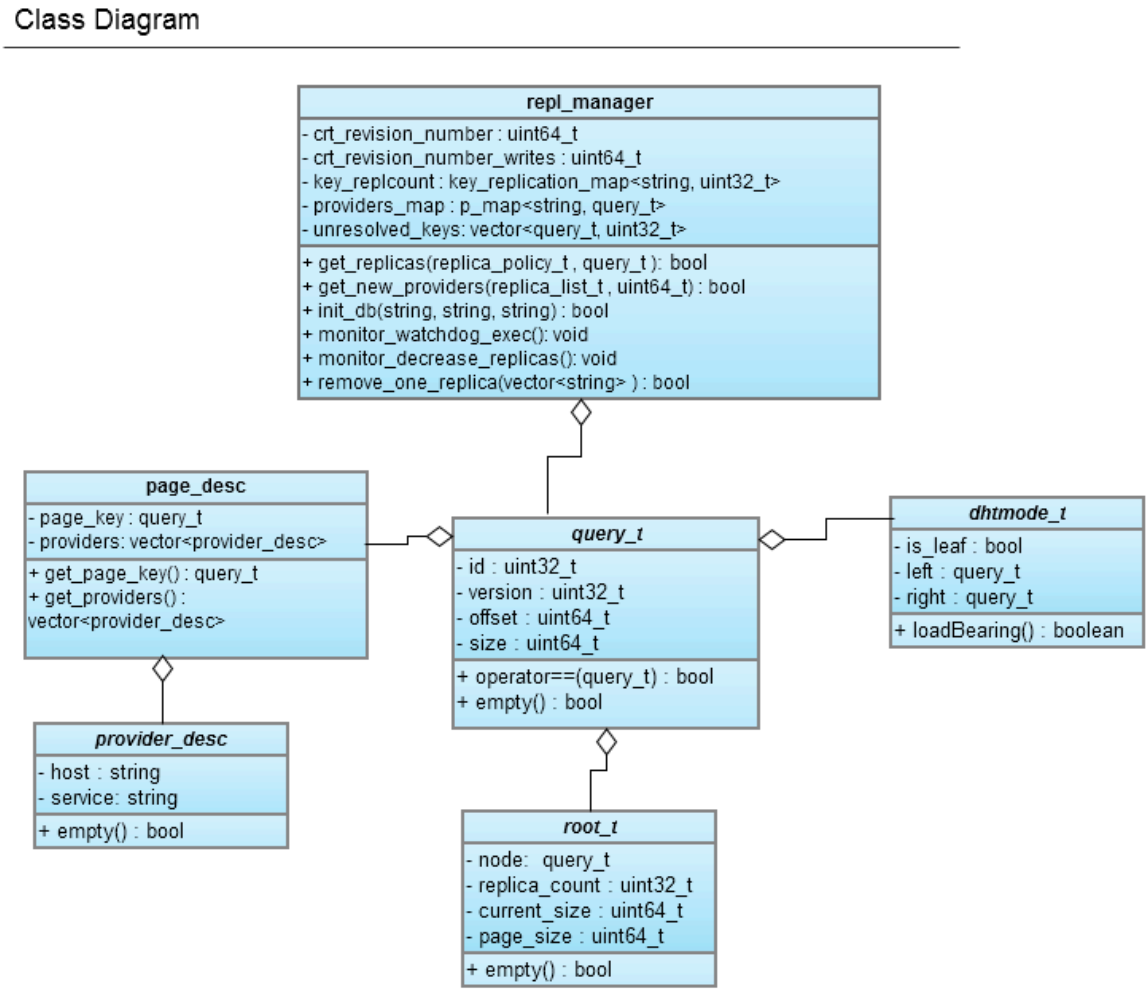


Figure 10. Replication Module – class diagram

The interactions diagram shown below comes to highlight the most important steps the Replication Manager makes during every interval. These steps respect the algorithms shown in the previous subchapter. Our air when designing this interactions diagram was to bring forth the way the system entities communicate with each other during every interval.

The first step is to communicate with the server database. This is not actually a BlobSeer entity, but, as we have stressed in the Appendixes, one cannot run our implementation without first ensuring there is a running PostgreSQL server in the system. During the next steps, the Replication Module also “discusses” with the Data and Metadata Providers.

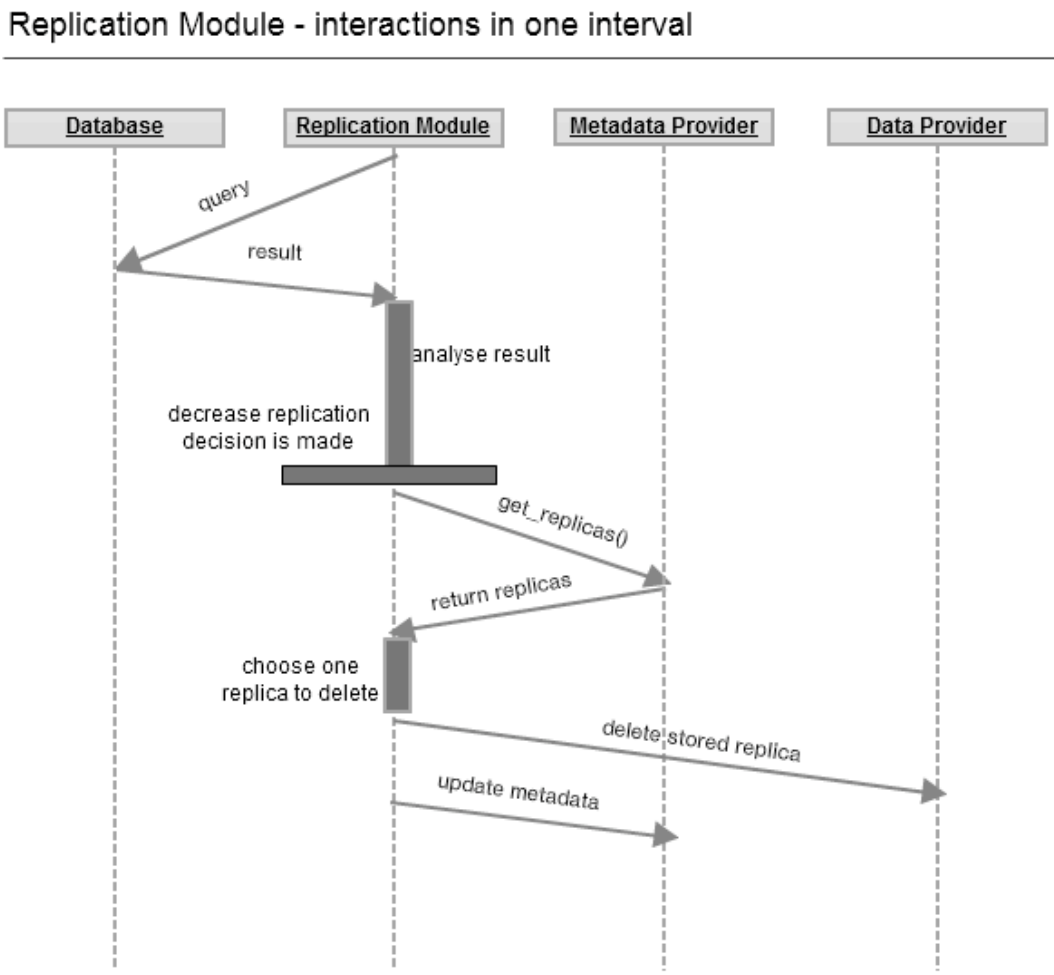


Figure 11. Replication Module - interactions

5.2 Scenario files

In this subchapter, we present in detail what the scenario files mean and show an example, which we then discuss. The scenario files make up the part of the Replication Module that puts all the self-adjustment capabilities of the system, and, therefore its entire power, at the reach of the client. By modifying various parameters, the client can control how flexible the system is. Here is an example of such a scenario file:

```
<scenario>
        <interval>30</interval>
         <load>0.2</load>
         <nr_of_reads_perinterval>7</nr_of_reads_perinterval >
```

```
        <desired_number_of_replicas>2</desired_number_of_replicas >
        <nr_of_writes_perinterval>3</nr_of_writes_perinterval >
</scenario>
```

The order in which the parameters are given is not important. However, at this stage of implementation of our Replication Module, these are the only available parameters that the client may use. Furthermore, the current implementation has another limitation which we are aware of: only one single scenario file is permitted per run, no matter how many blobs are created. In the future, we may considered altering the create primitive, in order to make it accept a scenario file per blob.

The most important parameter is the interval. It indicates the rate at which the module queries the database that contains the monitored information. This parameter is indicated in minutes, and there is no required range of its value.

Secondly, a very important role is played by the desired_number_of_replicas. No matter what happens in the system, the Replication Module is not allowed to decrease the replication factor lower than this threshold. It is allowed, on the other hand to further increase it if necessary. Therefore, this parameter can be seen as a "low limit" for the replication factor.

This is also the case when it comes to the nr_of_reads_perinterval and the nr_of_writes_perinterval. The situation is exactly the opposite for the load parameter, whose value can be interpreted as a "high limit". Any higher value than the given number can justify a decrease of the replication factor for the pages situated on the monitored Data Provider.

Another crucial aspect is the fact that all rules are taken into consideration when a decrease decision is made. This means that it is not enough for only one threshold to be exceeded. In order to make a decision which doesn't contravene with the requirements of the client , all the given thresholds must be exceeded. On the other hand, we must mention that the client is not forced to provide values for all these parameters. If, for instance the he/she is not interested in the number of writes that happen in a interval of time, this parameter can be left out of the scenario file. If no value is provided for the interval parameter, a default one is used.

5.3 Important data structures used

One key aspect we must point out is that, although the Replication Module interacts constantly with the other entities of the system, as was presented in the previous subchapter, it tends to not rely on the data they give, but to store and use its own personal findings. This is the case, for instance with the providers_map and the key_replcount. What happens exactly is that, as the Replication Manager runs, it collects information about everything else that goes on in the system. By storing locally this type of data, the Replication Manager brings forth two big advantages: reduced overhead, when trying to compute something based on these data, and avoidance of the single point of failure.

Another very important data structure is the unresolved key list, which is made up of keys – following the pattern: <id, version, offset, page_size> and values – the number of replicas needed in the system, but not yet created because of various reasons, such as lack of space of available Data Providers.

6. Experimental results

The Adaptive Data Replication Module that we modified and then included in the BlobSeer architecture is meant to deal with the problems of maintaining, increasing and decreasing the replication factor of blobs. We have concentrated our tests on the decrease part of the Replication Module, aiming to point out the way it modifies BlobSeer's performance.

The communication between the Replication Module and BlobSeer is achieved through Remote Procedure Call (RPC) messages using either TCP or UDP in order to send them across the network. Therefore, one of the main metrics used in evaluating our module is the overhead it induces to the system. The aim of such tests is to show exactly how intrusive the module we designed is.

Moreover, another important aspect to take into consideration is the delay required by our system to respond. That is to say, how much time it takes until the Replication Module decides to modify the replication factor. This, although it may not seem as a particularly important aspect, is in fact crucial for the cloud environment in which BlobSeer is meant to work, because, as you know, time is money as far as clouds are concerned.

6.1 Testing scenario

As a testing environment, we have used one or two machines, depending on the particular scenarios. The machines used are identical and have dual core processors, 2 GB of RAM and 20GB hard disks. These features where enough for our testing purposes, as we did not aim to include huge blobs in our testing scenarios. We have tested mainly with 256MB blobs, with a page size of 64KB. All the results presented in the next subchapter are obtained using these particular numbers. We have chosen these numbers, because we consider that they make excellent medium values, and have also been used in the sample programs provided in BlobSeer. However, we must mention that there were no striking differences when testing with other page or blob sizes.

Furthermore, we believe that scaling the results is also possible, as the rules still apply no matter how many blobs there are in the system. By rules, we understand the patterns and tendencies we have observed by comparing and contrasting the system's performance with and without our module included.

A typical used scenario was comprised of the creation of one or more blobs, writing data to these blobs and then developing a write-read pattern made to determine the creation of replicas in the system. Afterwards, the reads and writes dropped below the limits given by the client, and we were able to observe and make record of the performance of the system, while the replication factor was being decreased by the Replication Manager.

From this point forward we present some of our most important findings as far as test results are concerned. Every graph is explained in detail and all results are interpreted.

6.2 Test results

In this subchapter we present the results we have obtained from the performed tests and explain in detail what significance they bare to the entire system.

Our first concern was to observe the overhead introduced by using our Replication Module in BlobSeer. We did this by measuring the load5 parameter while running the tests mentioned in the previous chapter on both the same version of BlobSeer (release 0.4), with or without the Replication Module included. In Figure 12 below we have represented the obtained values.

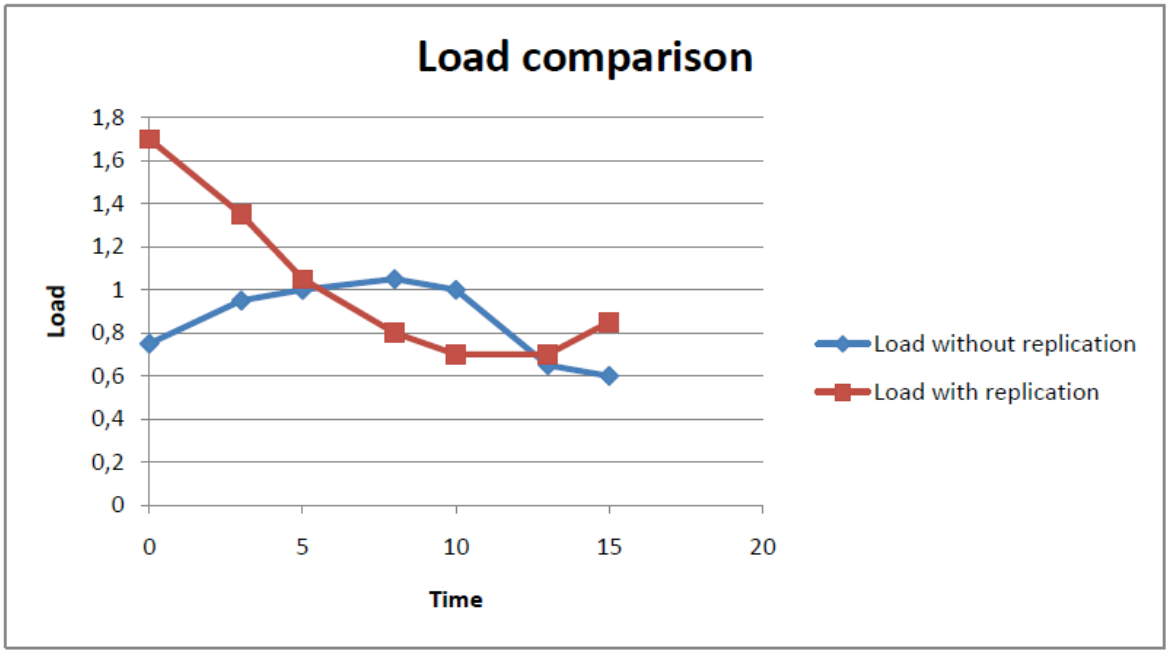


Figure 12. Load5 results

These results are not at all surprising, as it would have been expected to have a higher load for the release version with the included Replication Module. However, at a closer look, one can see that there is a big period of time for which the load5 parameter is actually lower with the Replication Module. This means that the module has achieved its goal and the replicas created in the system make access a lot easier and faster. The only times when the load is higher for our modified version of BlobSeer is when replicas are created or deleted from the system.

Our conclusions for this test are that, although the load is sometimes higher than in the unmodified version, it is at an acceptable value, thus showing the Replication Module is not very intrusive and does not affect dramatically the overall performance of the system. On the contrary, there are cases when it improves BlobSeer's performance.

The next parameter we took into consideration was the amount of free memory left in the system. This is actually a very powerful indicator of the intrusiveness and efficiency of our module. The results are shown in Figure 13 below.

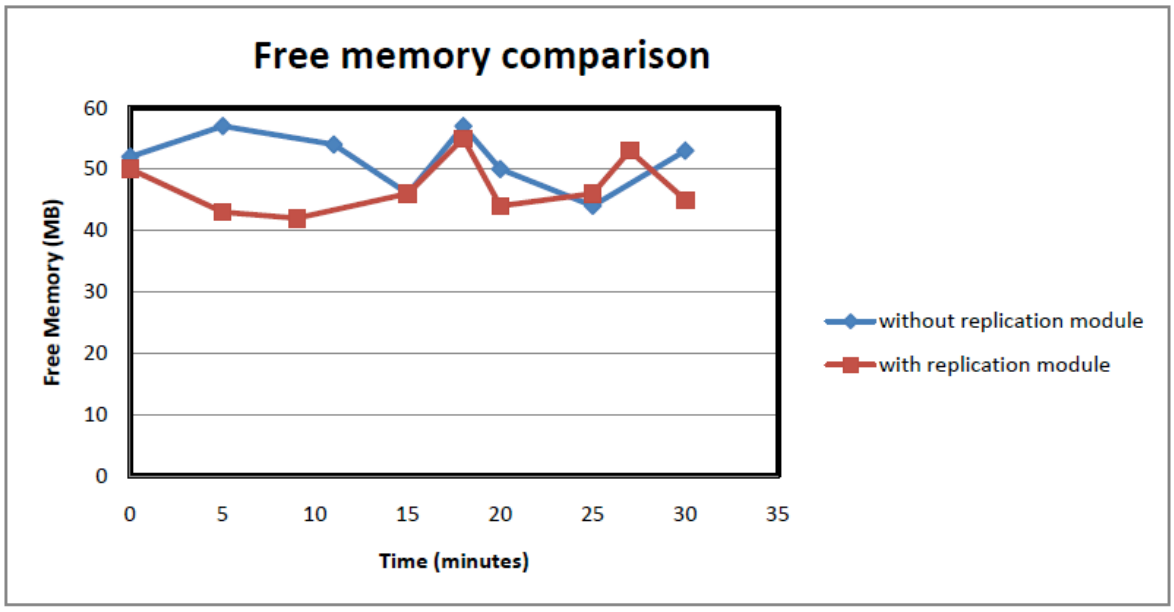


Figure 13. Comparative free memory results

As you can see, the average free memory is of 50MB. The fact that our module does not stray far from this figure is very important and not to be taken likely. It is another powerful indicator that recommends our module. However, we must observe the trends visible in both cases. With the Replication Module inserted, the tendency is to go a little below the average values, which means a little more memory used. This was to be expected, as there are more messages passed in the system and there is more stored data (the replicas created). In contrast, the average without the Replication Module tends to be a little higher. As an observation, we point out that there are situations, for example in the graph above, between the 0 and 15 minutes, when the free memory is lower. This is because, at first replicas are created. After that, the free memory increases with the deletion of replicas, reaching approximately the level of the one without replication. This means an important saving of space.

Next, we have analysed the occupied RAM memory in both cases. Figure 14 below shows the values without replication, while Figure 15 shows the values obtained with our module. At a closer look, we observer that the values obtained for this test were very much similar, pointing out that, although our module uses more physical memory, there are not many items stored in RAM. It is actually a characteristic of our module to keep as much information about the system stored on disk and to use the results from the monitoring databases, which are also stored on the disk, not in RAM.

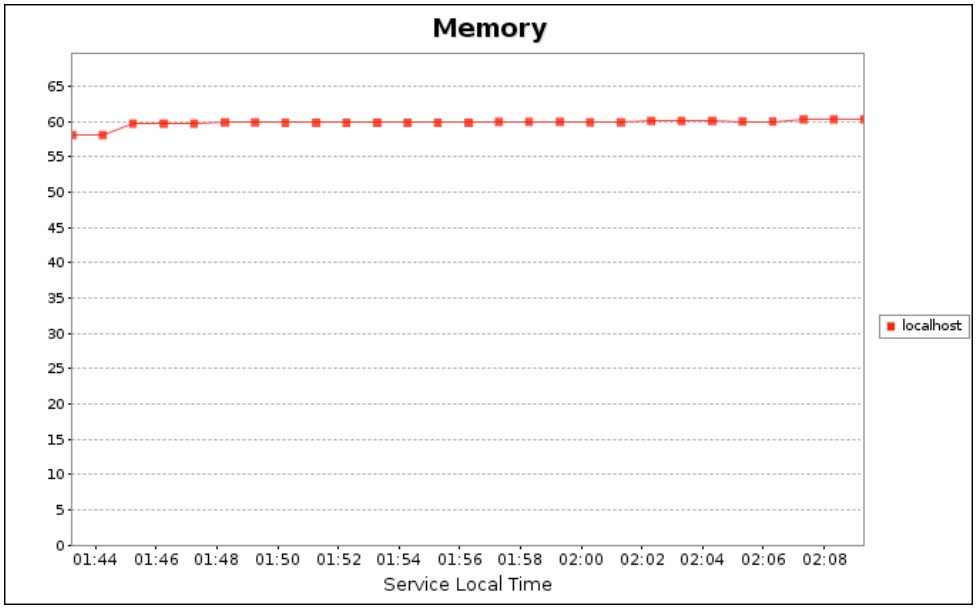


Figure 14. Occupied Memory – without replication

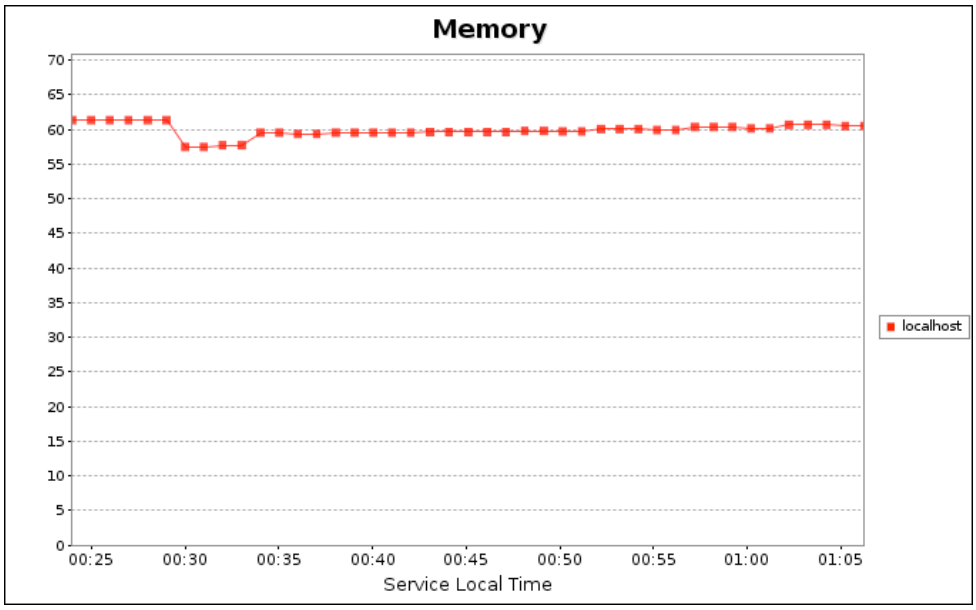


Figure 15. Occupied Memory – with replication

6.2 Performance evaluation

7. Conclusions and future work

7.1 Contribution

This thesis proposed our approach to enhance the Replication Module that already exists within BlobSeer, by adding the decrease replication factor functionality. By adding this capacity to

the Replication Module, the system thus became completely capable of dealing with the various cases of replication possible: increase, decrease or maintenance of the number of replicas for a given blob version.

This goal was achieved by using monitoring service and MonALISA repository. The monitored data were afterwards used to help the Replication Manager make an informed decision concerning the replication factor.

The fully functional Replication Module integrates perfectly with BlobSeer and helps it become an autonomic system regarding the managing and/or creating or deleting replicas of data stored on the Data Providers.

7.2 Future work

Although the adaptive data replication module, as it has been described in this paper, has all the necessary components to work properly and accomplish the purpose for which it has been designed, some improvements might be made to it.

A possible next step would be to optimise the DecisionMaker, in order to take various metrics into account when selecting the Data Provider from which we delete data. For instance, these metrics might include the Data Provider's position in the network, in order to reduce overhead, etc. Also, some advanced metrics might improve the precision of the Replication Module when altering blobs' replication factor. This would be a very important step, because more precision means less messages passed in the system and, overall, less overhead.

Another improvement, which comes to continue the whole distributed idea behind BlobSeer, would entail the distribution of the Replication Manager, thus creating more Replication Managers and eliminating the problem of single point of failure.

Acknowledgements

This bachelor thesis would not have been possible without the help, support, feedback and ideas of my supervisor, Dr. Ing. Alexandru Costan. His input has been invaluable, on both academic and personal level. I would also like to thank Alexandra Carpen – Amarie and Cătălin Leordeanu for their continuous help and collaboration.

References

[1] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," J. Parallel Distrib. Comput., vol. 71, pp. 169-184, 2011.

[2] Jeffrey O. Kephart, David M. Chess, "The Vision of Autonomic Computing", published by the IEEE Computer Society, January, 2003

[3] Lucian Cãncescu, Alexandru Costan, Valentin Cristea, “Self-adaptive Data Replication for BlobSeer using Monitoring Information”
[4] Thanasis Loukopoulos and Ishfaq Ahmad, “Static and Adaptive Data Replication Algorithms for Fast Information Access in Large Distributed Systems”, 2000, http://portal.acm.org/citation.cfm?id=851807
[5] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung., "The Google file system", ACM SIGOPS Operating Systems Review, Volume 37 , Issue 5, pp. 29-43, December, 2003
[6] The Hadoop Distributed File System. http://hadoop.apache.org/core/docs/current/hdfs_design.html
[7] Bogdan Nicolae, Gabriel Antoniu, Luc Bogue, “Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach”, 2009, http://blobseer.gforge.inria.fr/doku.php?id=main:publications
[8] BlobSeer: Alexandra Carpen-Amarie, Jing Cai, Alexandru Costan, Gabriel Antoniu, Luc Bougé, “Bringing Introspection Into the BlobSeer Data-Management System Using the MonALISA Distributed Monitoring Framework”
[9] Viet-Trung Tran, Gabriel Antoniu, Bogdan Nicolae, Luc Bougé. “Towards a Grid File System Based on a Large-Scalable Blob Management Service”, published in "CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service computing (2009)"
[10] Badiu Mihaela, Cătălin Leordeanu, Valentin Cristea, “BlobSeer False Data Deletion”, 2011
[11] B. Nicolae, G. Antoniu, and L. Bougé, "BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency," in Proc. EDBT/ICDT '09 Workshops, Saint-Petersburg, Russia, 2009, pp. 18-25
[12] The MonALISA Project: http://monalisa.caltech.edu/monalisa__Publications__Papers.htm
[13] http://monalisa.caltech.edu/docs/marco_thesis.pdf
[14] Mihaela Vlad, “Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System”, 2010

Appendixes

A1 Minimum system requirements

In order to be able to properly compile and run our modified version of BlobSeer, you need to have the following items listed below installed. Some of them are libraries, others are full applications. As far as we can tell, all of them are required, none of them being optional to the correct functioning of the system. We mention that the settings below are fully compatible with Ubuntu 10.9 on which the tests have been carried out:

- libpq-dev

- libpqxx-3.0 a simple apt-get install will do
- libpqxx-dev
- gmake: sudo ln –s /usr/bin/make /usr/bin/gmake
- readline: sudo apt-get install libreadline6-dev
- zlib: source installation from http://www.zlib.net/zlib-1.2.5.tar.gz
- PostgreSQL source installation – version postgresql-9.0.4.tar.gz

  Warning: If you have a Debian or Ubuntu environment, you should take into consideration that the system looks for the started postgresql server by default in /var/run/postgresql. On the other hand, the operating systems mentioned above look by default in the /tmp directory. This discrepancy generally results in an inability of applications to communicate with the database server (even though it is running). The solution for this type of problem is to correctly set the unix_socket_directory in the postgresql.conf file, which is found, in our case, in /home/alex/MonaLisa/Service/myFarm/pgsql/data.

  unix_socket_directory = '/var/run/postgresql'
- ApMon – download from :

  http://monalisa.cacr.caltech.edu/download/apmon/ApMon_cpp-2.2.6.tar.gz

  and follow these steps:

  ./configure --prefix=$HOME/deploy

  make

  make install

  - Also install the Java library from here:

    http://monalisa.cacr.caltech.edu/download/apmon/ApMon_java-2.2.4.tar.gz

    and follow these steps:

    set the JAVA_HOME environment variable;

    run the installation scripp: ./build_apmon.sh
- MonALISA
  - Installation from http://monalisa.cern.ch/monalisa__Download__ml_license.html
  - Click REGISTER ACCEPTANCE
  - Unzip the archive and enter the newly created folder
  - Run install.sh – during this step you will be required to choose a name for your farm and fill in some other information; also you must enable ApMon support – the port used by ApMon is, by default, 8884)
  - Enter the installation folder (by default: ~/MonaLisa) and then the subfolder Service/myFarm
  - Edit the ml.properties file, in order to set the IP for the machine (modify the lia.Monitor.useIPaddress property)
  - In order to start the MonALISA Service, enter the CMD folder and execute "./ML_SER start". When you are done using the service, execute "./ML_SER stop". If in doubt that the system is working properly, check the log files found in the Service/myFarm subfolder(e.g.: ML0.log,etc)
  - Optionally, you may choose to install the MonALISA interactive client, in order to have a GUI for the monitored parameters. In order to install it, you simply download and run the jnlp file found at this location: http://monalisa.cern.ch/mlclient/Monalisa.jnlp

- Download the monitoringDB folder from the INRIA svn: svn://scm.gforge.inria.fr/svn/blobseer/contrib/monitoringBlobSeer/monitoringDB, using the svn checkout command.
  Make sure to follow the exact instructions found in the README file and compile the program. Afterwards, modify the files found in the conf directory. In the AppDb.properties file fill in the exact information about the database in which your monitored data will be stored (database name, port, etc.).
  Warning: Edit the ml.properties file, in order to point to the class files corresponding to the filters you want to apply. Also add a property as follows to point to the node file:
  BSMonFilter.nodeFile=file:/home/alex/monitoringDB/conf/nodeFile.txt

A2 Steps required when running the application

1. Start postgres :
   alex@ubuntu:~$ /usr/local/pgsql/bin/postmaster –D ~/MonaLisa/Service/myFarm/pgsql/data/
   One of the most commonly encountered errors with postgresql is the impossibility to find a stable recovery point. This generally happens when you restart the postgresql database server. Do not panic, there is a simple solution to this: use the pg_resetxlog command as shown below. The –f option means forcing the server to redo its logs and be able to start again. Do not worry about loss of data, as the required databases are dynamically generated every time BlobSeer is run.
   alex@ubuntu:/usr/local/pgsql/bin$ ./pg_resetxlog -f /home/alex/MonaLisa/Service/myFarm/pgsql/data
2. Manually create the blobseer_monitor and mydb databases, using the created command.
3. Start the MonaLisa monitoring Service:
   alex@ubuntu:~/MonaLisa/Service/CMD$ ./ML_SER start
   Check the log files in order to make sure everything is in order (ML0.log) found in …/Service/myFarm.
4. Start the monitoring thread:
   alex@ubuntu:~/monitoringDB$ java -classpath $DM_PROJECT_HOME/lib/log4j-1.2.15.jar:$DM_PROJECT_HOME/lib/postgresql-8.4-701.jdbc4.jar:$DM_PROJECT_HOME/lib/MSRC_WEB.jar:$DM_PROJECT_HOME/lib/FarmMonitor.jar:$DM_PROJECT_HOME/build/:. processing.NodeApp ~/MonaLisa/Service/myFarm/ml.properties 5 5000
   Remember to set the $DM_PROJECT_HOME first:
   export DM_PROJECT_HOME=/home/alex/monitoringDB
5. Start the application itself(mind the order of the processes): vmanager, pmanager, monitor, rmanager, sdth and provider
6. You can also manually check the databases, to see the information registered in them. Use the following postgresql commands:
   - see database: psql <database_name>

- see tables in a database: \d or \dp
- see columns in a table: \d table_name
- see all rows: select * from table_name;

A3 Sample configuration file

In our paper, the configuration files used to run the BlobSeer processes are encountered in the form of .cfg files. Here is a sample file for a simple run, including one instance of each process BlobSeer is made up of:

```
# Version manager configuration
vmanager: {
   # The host name of the version manager
   host = "localhost";
   # The name of the service (tcp port number) to listen to
   service = "2222";
};

# Provider manager configuration
pmanager: {
   host = "localhost";
   service = "1111";
};

# Replication manager configuration
rmanager: {
   host = "localhost";
   service = "3333";
   mlrepo = "/home/alex/release-0.4/rmanager/mlrepo.txt";
   mlservice = "/home/alex/release-0.4/rmanager/mlservice.txt";
};

# Monitor configuration
monitor: {
   host = "127.0.0.1";
   service = "4444";
};

# Provider configuration
provider: {
   service = "1235";
   # Maximal number of pages to be cached
   cacheslots = 1;
```

```
    # Update rate: when reaching this number of updates report to provider manager
    urate = 100;
    dbname = "/tmp/blobseer/provider/db/provider.db";
    #dbname = "";
    # Total space available to store pages, in MB (1GB here)
    space = 128;
    # How often (in secs) to sync stored pages
    sync = 100;
    # How many times to retry writing from client side before giving up
    retry = 1;
    # Use compression?
    compression = true;
};

# Built in DHT service configuration
sdht: {
    # Maximal number of hash values to be cached
    cacheslots = 1000000;
    # No persistency: just store in RAM
    dbname = "";
    # Total space available to store hash values, in MB (128MB here)
    space = 128;
    # How often (in secs) to sync stored hash values
    sync = 10;
    # Use compression?
    compression = false;
};

# Client side DHT access interface configuration
dht: {
    # The service name of the DHT service (currently tcp port number the provider listens to)
    service = "1234";
    # List of machines running the built-in dht (sdht)
    gateways = (
        "localhost"
    );
    # How many replicas to store for each metadata entry
    replication = 1;
    # How many seconds to wait for response
    timeout = 10;
    # How big the client's cache for dht entries is
    cachesize = 1048576;
};
```

Most settings are the ones suggested in the tutorial provided from INRIA:

http://blobseer.gforge.inria.fr/doku.php?id=tutorial:deployment. We have, however, also added proper settings for the other two actors in the system: Monitor and Replication Manager. As a side note, when configuring the rmanager make sure that the paths for the mlrepo and mlservice files are correctly given. Otherwise, you will get an error and will be unable to continue running the application.

If you decide to run more than one provider, which is usually the case, as the system has been designed for a large number of providers that enter and leave the system at will, you must create more configuration files, resembling the one shown above. However, you must make sure that the ports given for the different providers are not the same. Also, you must include for every provider, or any other instance for that matter, the configuration settings for all the others. That is to say, even if only the setting for the provider changes, in the second configuration file you must also include the settings referring to the Version Manager, for example.