Universitatea “Politehnica” Bucureşti
Facultatea de Automatică şi Calculatoare, Catedra de Calculatoare

Self-adaptive Data Replication for BlobSeer using Monitoring Information

Coordonatori Ştiinţifici:
Prof. Dr. Ing. Valentin Cristea
As. Drd. Ing. Alexandru Costan

Autor:
Lucian Cãncescu

Bucureşti, 2010

Abstract

Adaptive Data replication for data storage systems and distributed computing represents an important element for autonomic behavior. It improves their overall performance when offering services such as IaaS on a cloud as it minimizes the storage resources used on large-scale distributed systems. Moreover, it increases the availability of highly accessed data. Depending on the type of changes that occur (e.g., modifications of the load, the type of workload, the available resources, etc.), different adjustments have to be made. We describe a technique to enable adaptive data replication within the BlobSeer distributed storage system. We discuss the system's architecture, present the replication algorithms that we rely on, and evaluate our tool within a distributed testing environment. The intended implementation is for C++.

Keywords – adaptive data replication, BlobSeer, MonALISA, storage, autonomic behavior, replication factor.

Table of Contents

1. Introduction

Distributed computing as well as clouds gain popularity for the services they offer. Most of them are used for their data processing power as well as storage facilities. Raw and processed data is stored on them for analysis and further processing. That is why data availability on such systems is essential for their overall functionality. Because of the nature of files that are increasing in size, it is often impossible to store a whole file on a single disk. Therefore, files have to be split in chunks that are distributed among data storage servers on networks.
If there happens to be a problem with a storage disk, then multiple files are affected. This problem can be solved by creating many copies of the chunks on other storage disks. The easiest way is to specify manually the number of desired copies for each file. In this situation, because the number of copies is specified manually at the file's creation, no further adjustments can be made. This proves to be a waste of space and ultimately a waste of money because disks cost and services' price depends on how much of storage resources are used. Thus, data storage systems and distributed computing networks have to be equipped with an adaptive data replication module. The adaptive data replication module is that part of a storage system that handles the replication problems and is able to maintain and adjust the replication factor for files they host in order to increase the reliability of the entire system. A good approach to implement such an adaptive data replication module is by accomplishing the services on the background, so that the end-user does not notice that there is any kind of problem with the storage disks. The difficulty of creating such a module consists in monitoring storage providers and the problems that may arise that can have different sources (networking issues, disk problems, uneven load on the storage disks, etc). All the causes are treated and decisions have to be made on how to deal with the chunks affected by the storage disks were problems were reported.

Data storage services such as BlobSeer[1] need an adaptive data replication module to handle the reliability issue. It is crucial that storage systems automatically adapt to changes that occur and provide a reliable access to content. The replication module, listens to the changes, evaluates what data chunks are more accessed by reads and writes and makes decisions of whether to increase, decrease or maintain the replication factor of a file.
Currently, BlobSeer is only able to create a fixed number of replicas on files' creation. However, the number of replicas is not maintained, nor increased or decreased based on real-time statistics for files' use. Therefore, a module to overcome these limitations was created. Our module is implemented for BlobSeer (§3.1) data storage service. The module helps BlobSeer adjust its replication scheme for each blob individually based on data collected from monitoring services such as MonALISA(§3.2). The fact that each file's replication factor is automatically adjusted implies lower costs for storage services on the cloud where the price depends on how much of resources are used.
In this paper we introduce the notion of adaptive data replication, which enables data storage services adapt automatically the replication scheme for the stored files. We describe various replication algorithms and emphasize the main advantages that our module adds to BlobSeer.

1.1 Thesis Objectives

The main purpose of this paper is to present the design and the implementation of an adaptive data replication module that makes the transition from static to dynamic replication. This module is part of a bigger project called BlobSeer which aims to provide a data storage service designed to deal with the requirements of large-scale data-intensive distributed applications.

In the current implementation of BlobSeer, the replication factor for each file is specified at the file's creation and remains unchanged through the file's lifetime. Even worse, if a storage provider that holds parts of that file fails, there is no mechanism to prevent that or to try to maintain that file's replication factor.

With the Replication Module, we have added to BlobSeer features that eliminate these deficits. Moreover, we added the support to automatically increase and decrease the replication factor for files depending on various parameters. The Replication Module enables BlobSeer to continue operating properly and offering access to data in the event of failure of one or many of its data storage providers.

Our goal was to create a Replication Module that solves the above-mentioned problems and integrates in an optimal way with BlobSeer.

First, there have been many challenges such as how to maintain the replication factor for the stored files and then, how to make decisions when to increase or decrease their replication factor. Since the beginning of the project, our goal was to create a module that operates fast whenever action needs to be taken and solves the issues in as less time as possible.

Second, we tried to achieve this by exchanging a minimum number of messages with BlobSeer. The Replication Module has to deal with data providers in those critical moments when they fail, become available or periodically update with information containing the written pages. Because speed is crucial to BlobSeer, a large number of exchanged messages would slow down the performance of the Replication Module. That is why we designed the module to communicate with BlobSeer exchanging messages only when needed.

Above all, the overhead that this module brings to the system needs to be reduced. The Replication Module computes locally the topology of BlobSeer based on the messages it receives from it. The availability of a local topology of providers with the pages they store brings advantages of reduced overhead as well as speed increase because all the necessarily information is already there. All the information that is locally stored is designed to be as compact as possible to save time on searches. With these in mind, we have started to design and implement the Replication Module for BlobSeer.

Finally, the project was divided in two important milestones. The first milestone consisted in maintaining the replication factor for stored files in case data storage providers become unavailable. We made the assumption that data storage providers failure, are the norm rather than the exception. This milestone has made BlobSeer fault-tolerant in the event of failure of storage providers. The second milestone brings BlobSeer the advantage of automatically increasing and decreasing the replication factor for the files it stores. With the help of monitoring services such as MonALISA Service and MonALISA Repository, we have managed to collect data regarding the use of blobs in terms of reads and writes. We have used these monitored data to compute metrics that allowed us to make replication factor decisions. Moreover, the Replication Module collects monitored data from MonALISA Repository regarding the load on storage providers. This brings us a double advantage: it is able to prevent a provider failure by creating in time copies of the data chunks it contains and it can decrease the load on the providers in case.

1.2 Thesis Outline

The reminder of this thesis is organized as follows:

Chapter 2 presents a survey of the current research efforts that deal with fault-tolerance and data replication problems. It includes several algorithms and the description of two file systems that already provide such mechanisms. For each algorithm and file system presented we included a short comparison against BlobSeer to show the advantages and disadvantages such an implementation would have on our data storage service.

Chapter 3 provides background information for BlobSeer, the data storage service that our replication module will be implemented for. The monitoring services that our module uses are handled by MonALISA. This chapter introduces basic information about how BlobSeer works, and describes the parts of MonALISA that are useful to our implementation, namely MonALISA Service and MonALISA Repository.

Chapter 4 presents the replication module's architecture. The concepts of Listener and Replication Manager are introduced as well as detailed information about maintaining, increasing and decreasing the replication factor. Moreover, this chapter presents the interactions between the Replication Module and BlobSeer components.

Chapter 5 outlines illustrations for various functionalities that the Replication Module brings to BlobSeer. There are scenarios for maintaining the replication factor, for increasing and decreasing it and how the module deals with busy providers that show increasing load levels.

Chapter 6 focuses on the implementation of our Replication Module. It gives detailed information for every component illustrated in chapter 4, how the communication with BlobSeer has been done and how we structured the work on two milestones, one for maintaining the replication factor and the other for increasing and decreasing it.

In Chapter 7, we evaluate our work. We include the results of several tests to outline the overhead that the Replication Module brings to BlobSeer. Moreover, we did an analysis on how the number of messages increases when the number of blobs becomes larger.

The last chapter of this thesis, Chapter 8, presents the conclusions on the Replication Module, based on the results obtained in last chapter's tests. Finally, we mention possible improvements for the Replication Module and outline the conclusions of this paper.

2. Related Work

Creating replicas of frequently accessed data chunks across a read-intensive network can result in large bandwidth savings which can lead to reduction in user response time. However, data replication in the presence of writes incurs an extra cost due to multiple updates. The set of providers at which an object is replicated constitutes its replication scheme. Finding an optimal replication scheme that minimizes the amount of network traffic, given the read, write and append frequencies for various objects, is NP-complete in general because of their fluctuation.
There are several algorithms that deal with the replication problem as well as file systems which already provide support for autonomic data replication system.
Below we will present a couple of the algorithms and we compare them to our adaptive data replication module.

2.1 SRA, AGRA

The paper by Thanasis Loukopoulos and Ishfaq Ahmad [2] presents several algorithms which deal with the *Data Replication Problem, DRP*. The first solution is a *static algorithm*, called *SRA*. It is efficient mainly with systems that are read-oriented. Because the static solution does not scale to the increasing number of writes, they proposed an *adaptive genetic algorithm,* called *AGRA* that overcomes the limitations of the SRA. A couple of elements that are worth following are data access time and data transfer cost. The latter algorithm proves to be efficient because it can rapidly adapt to the dynamically changing characteristics such as the frequency of the number of reads and writes.
The SRA static algorithm is based on a greedy method. It computes a benefit value per storage unit. This will help the system determine whether an object will be replicated on a certain site. Note that the algorithm yields good solutions when the number of reads is large.
Each site on the network will send during defined large periods of times (for example, every night) the read – write patterns for the objects the system hosts. Thus a central processing unit, called *the monitor* will determine where each object should be replicated in order to tune its access time and its network transfer cost. This proves to be inefficiently if the number of reads and writes fluctuates (for example, during day time).
In order to overcome the above mentioned limitations, the AGRA is a dynamic algorithm that makes an analogy between the initial replication scheme of the objects and an initial population of chromosomes. It works based on the three genetic search operations, namely, selection, crossover and mutation. In this case, a chromosome will contain all the objects and each bit that it contains will show the sites where they are replicated. In order to tune the replication scheme, each time the read – write pattern for an object changes below a threshold, the AGRA algorithm is executed to determine its new replication scheme.

2.1.1 Implementation details for SRA

To present the SRA algorithm, we maintain a list $L^{(i)}$ for each site i, $S^{(i)}$ containing all the objects that can be replicated. An object Ok can be replicated at site i, $S^{(i)}$, only if the remaining storage capacity $b^{(i)}$ of the site is greater than its size (i.e. $b^{(i)} \geq$ ok) and the benefit value is positive. The benefit value $Bk^{(i)}$ consists of the expected benefits in NTC (network transfer cost) if we replicated O_k at $S^{(i)}$. The benefit is computed by using the difference between the NTC occurred from the current read requests, which would be eliminated if we made a replica, and the NTC arising due to updates to that replica. The algorithm keeps a list LS containing all the sites that have the opportunity to replicate an object. A site $S^{(i)}$ is in LS only if $L^{(i)}$ is not null. The SRA performs in steps. In each step, a site $S^{(i)}$ is chosen

from LS in a round-robin fashion and the benefit values of all objects belonging to $L^{(i)}$ are computed. The one with the highest benefit value is replicated and the lists LS, $L^{(i)}$, together with the corresponding nearest site value $SN_k^{(i)}$, are updated accordingly.
In this case, on each step there will be O(M+N) operations to execute, where N is the number of objects in the distributed system and M is the number of sites in the network. In the worst case, where each site has enough free space to hold all the objects and the number of updates is 0, there are MN such iterations. Hence, the complexity of the SRA algorithm is $O(M^2N+MN^2)$.

2.1.2 Implementation details for AGRA

Each chromosome in the population of AGRA is a bitstring of length M. The O_k is the object for which the algorithm is run. A 1 value in the *i*th bit of it denotes that site i, S(i), holds a replica of Ok. In the algorithm Ap will represent the population size and Ag the number of generations it evolves. The initialization of the first generation is performed by randomly generating half of the population while the rest is obtained from the solutions previously found by the static genetic algorithm. The three operations of GA (Selection, Crossover, Mutuation) come afterwards to define the population of the next generation. The V_k will deonte the NTC occured due to reads and updates of the object O_k. Vk can be computed by omitting the summation for all objects 1 .. N in the total data transfer cost function, D.
After A_g generations, AGRA terminates having converged largely to find solutions in distributing Ok to a degree that minimizes the NTC of R/W requests. The best R_k found by AGRA is copied to half of the initial population of the genetic algorithm, including the corresponding elite chromosome (current network replica distribution), while the remaining R_ks are randomly copied to the other half. Such transfer can result in storage capacity violation which needs to be resolved efficiently. Other than randomly de-allocating objects until the constraint is satisfied, we can follow a greedy method and calculate the negative impact each possible one object de-allocation has in the total data transfer cost function, D value. This can be $O(M^2N)$ in the worst case, where N stands for the number of objects in the distributed system and M for the number of sites in the network.
In order to estimate how beneficial a replica is (from the NTC reduction perspective), we must take into consideration both global properties of the object and local characteristics. 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 a high update ratio, but being widely distributed will have more chances of being selected for de-allocation. A special weight to the local read requests needs to be given. Large objects with poor local read demand are preferred for de-allocation, since this will result in freeing up more space for future allocation. Finally, we include an estimation of how good the site acts as the potential closest neighbor of other sites, by calculating its proportional link weights. The computational complexity of the estimation is O(M) for a single object, where M stands for the number of sites in the network.

The replication module for BlobSeer meets the adaptive algorithm requirements but, unlike it, it is able to automatically adjust to the provider failures and maintain the replication scheme.

2.2 ADR

The paper by Ouri Wolfson, Sushil Jajodia, Yixiu Huang [3] presents an algorithm for dynamic replication of objects in distributed systems. The algorithm is called *Adaptive Data Replication, ADR* and works only on a tree network (no redundancy loops). It is adaptive in the sense that it changes the replication scheme of the object (i.e. the set of processors at which the object is replicated) as changes

occur in the read – write pattern of an object (i.e. the number of reads and writes issued by each processor). The algorithm continuously moves the replication scheme towards an optimal one. The changes in the read – write pattern may not be known priori.
The algorithm is distributed. This means that each processor makes decisions to locally change the replication scheme and it does so based on statistics stored locally. Distributed algorithms present the following two advantages: they respond to changes in a more timely matter and their overhead is lower because they eliminate the extra messages required in the centralized case.
ADR changes the replication scheme to decrease its communication cost. The communication cost of a replication is the average number of interprocessor messages required for a read or a write of the object.
In the ADR, the initial replication scheme consists of a connected set of processors. At any time, the set of processors that form the replication scheme must be connected. The replication scheme expands as the read activity increases, and it contracts as the write activity increases. If the number of reads equals the number of writes, it remains fixed.
The ADR algorithm services the reads and writes of an object. They are executed as follows. A read of the object is performed from the closest replica in the network. A write updates all the replicas, and it is propagated along the edges of a subtree that contains the writer and the processors of the replication scheme.
The replication scheme changes dynamically every time period. The need for changes is determined using three tests, namely, the expansion test, the contraction test and the switch test.

2.2.1 Implementation details for ADR

In the ADR algorithm, the processors of the replication scheme, denoted R, are connected. A Ŕ-neighbor is that node that belongs to R but has a neighbor that does not belong to R.
Each processor executing the ADR algorithm receives a priori a parameter *t* denoting the length of a time period (e.g., 5 minutes). Changes to the replication scheme are executed at the end of the time period, by some processors of the replication scheme. The need for changes is determined using three tests, namely, the expansion test, the contraction test, and the switch test. The expansion test is executed by each processor that is an Ŕ-neighbor. Suppose that R is not a singleton set. Then we define a R-fringe processor to be a leaf of the subgraf induced by R. each R-fringe processor executes the contraction test. A processor can be both an Ŕ-neighbor and R-fringe. Then it first executes the expansion test, and if it fails, then it executes t he contraction test. A processor p of R that does not have any neighbors that are also in R (e.g. R is the singleton set {p}) executes first the expansion test, and if it fails, then it executes the switch test.
Expansion Test. For each neighbor j that is not in R, compare two integers denoted x and y. x is the number of reads that i received from j during the last time period; y is the total number of writes that i received in the last time period from i itself, or from a neighbor other than j. If x > y, then i sends to j a copy of the object with an indication to save the copy in its local database. Thus j joins R. Except from i and j, no other processor is informed of the expansion of R. The expansion test is performed by comparing the counters (one for the reads and the other for the writes). The counters are initialized to zero at the end of each time period and incremented during the following time period. The expansion test succeeds if the (if) condition is satisfied for at least one neighbor. It fails if it does not succeed.
Contraction Test. Compare two integers denoted x and y. x is the number of writes that i received from j during the last time period; y is the number of reads that i received in the last time period (the read requests received by i are made by i itself or receivfed from a neighbor of i different from j). If x>y, then i requests permission from j to exit R, that is, to cease keeping a copy. If there are only two

processors that hold a copy, the one that decides to cease the copy, announces the other processor his intention.
If processor i constitutes the entire replication scheme, then it is a Ŕ-neighbor; thus it must execute the expansion test. If the expansion test fails, then i executes the following (switch) test at the end of the time period.
Switch Test. For each neighbor n, compare two integers denoted x and y. x is the number of requests received by i from n during the last time period. If x>y, then i sends a copy of the object to n with an indication that n becomes the new singleton processor in the replicationscheme, and i discards its own copy.
When the (if) condition of the contraction or switch test is satisfied, then we say that the test succeeds, otherwise it fails.

At the end of each time period, an Ŕ-neighbor executes the expansion test. An R-fringe processor executes the contraction test. A processor that is both an Ŕ-neighbor and an R-fringe, executes first the expansion test and, if it fails, then it executes the contraction test. A processor of R that does not have a neighbor in R, executes first the expansion test, and, if it fails, then it executes the switch test.

Because on BlobSeer, a blob's data chunks are distributed among the providers the replication module will communicate with the provider manager. Although the network can be seen as a tree with the provider manager the root and the providers the leaves, the replication scheme will change only based on the read - write operations. Unlike ADR where a processor contains the full file, on BlobSeer, a provider contains only several chunks of it. Therefore it is unnecessarily to move the replication scheme closer to the source of reads because the time will not decrease. As it was mentioned, it communicates with the provider manager and makes decisions on how to detect a dead provider, how to back-up data from it and how to increase or decrease the replication scheme. All these make the decisions less time-consuming for BlobSeer.

2.3 Dynamic Data Replication on various file systems

Among the file systems that provide dynamic data replication we will briefly mention two of them, namely Google File System (GFS) [4] and Hadoop Distributed File System (HDFS) [5].

2.3.1 The Google File System

In the first case, GFS [4] deals with the dynamic replication problem as follows. By default it stores three replicas for each file, but users can specify other replication levels for different parts of the file namespace. The replication decisions are made by the master. It re-replicates chunks of files as soon as the number of available replicas falls below a user-specified goal. Each chunk that needs to be re-replicated is prioritized based on several factors such as how far it is from its replication goal. The placement of a new replica is based on equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks.
The master rebalances replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Another attribute is to manage replicas in order to keep them up-to-date, such as when the chunk server goes up and needs to be synchronized to the last version.
This paragraph explains how GFS places a chunk replica. A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunk-servers spread across many machine racks. The

chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. Chunk replicas are spread across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline. It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks – a tradeoff.
The more detailed process of re-replication is the following. The master re-replicates a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, a chunk that has lost two replicas receives a higher priority than one that has lost only one replica. The systems prefer to replicate chunks for live files as opposed to chunks that belong to recently deleted files. Moreover, to minimize the impact of failures on running applications, the system boosts the priority of any chunk that is blocking client progress.
The master picks the highest priority chunk and replicates it by instructing a chunk-server to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active replication operations on any single chunk-server, and spreading replicas across racks. To keep replication traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunk-server. Each chunk-server limits the amount of bandwidth it spends on each replica operation by throttling its read requests to the source chunk-server.
The master rebalances replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunk-server rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunk-servers with below-average free space so as to equalize disk space usage.

On BlobSeer, the replication module is notified when a provider stops responding. When that happens, the Replication Manager already knows what chunks were held on it and tries to create copies of them on the remaining providers. Thus, the replication scheme is preserved.

2.3.2 HDFS

HDFS [5] is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks. All blocks in a file, except the last block, are the same size. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.
The NameNode makes all decisions regarding replicaiton of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A blockreport contains a list of all blocks on a DataNode.
HDFS's reliability and performance relies mainly on the placement of replicas. Optimizing replica placement, improves data reliability, availability, and network bandwidth utilization.
For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a different node in the local rack, and the last on a

different node in a different rack. This policy cuts the inter-rack write traffic which generally improves write performance. Note that the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across remaining racks. This policy improves write performance without compromising data reliability or read performance.
When a read is performed, HDFS tries to satisfy the request from a replica that is closest to the reader. Thus the global bandwidth consumption and read latency are minimized. If there exists a replica on the same rack as the reader node, then that replica is preferred to satisfy the read request. If the HDFS cluster spans multiple data centers, then the replica that is hosted on the local data center is preferred over any remote replica.
HDFS provides a *Safemode* state. On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks in with the NameNode, the NameNode exits the Safemode state. It then determines the lista of data blocks that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes.

As for BlobSeer, the replication factor for a blob is specified on file creation. Based on the read / write fluctuations, it can increase or decrease. This helps the storage service to free unused space and fill it with copies of the highly used blobs.

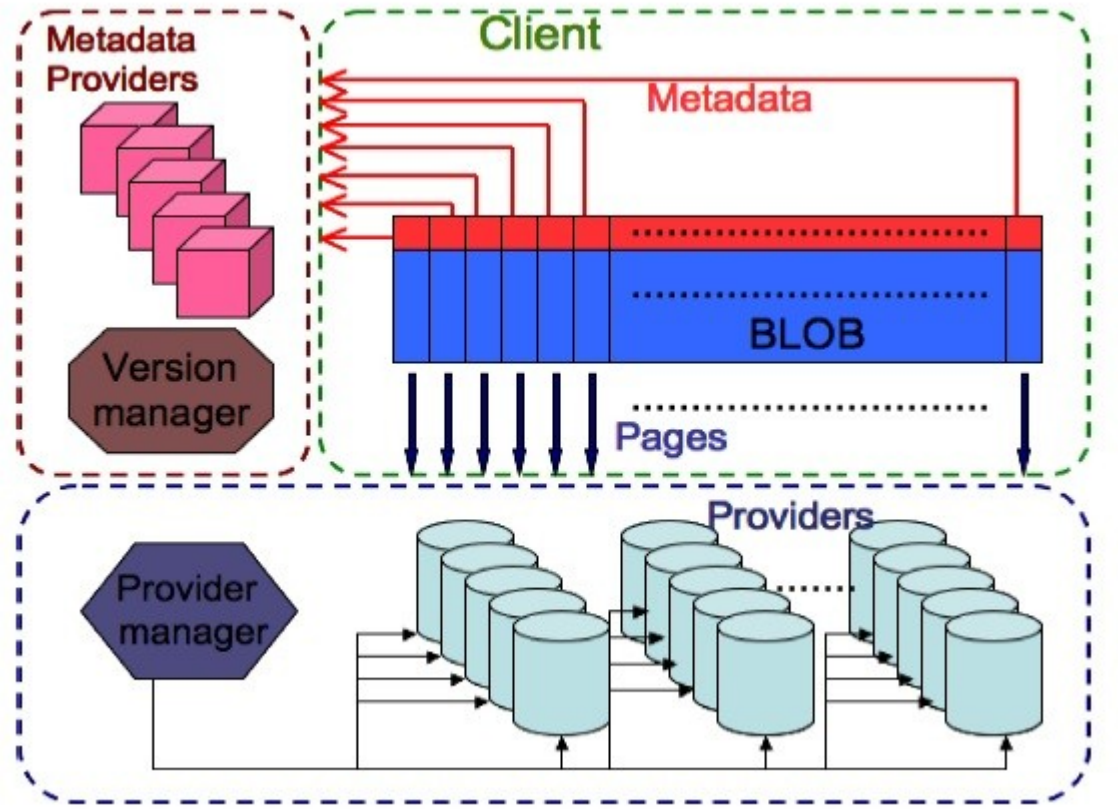
The Replication Module on BlobSeer, unlike the above mentioned replication algorithms and implementations, has an up-to-date view of the data stored by the providers. It knows at any time what chunks are on each data provider. Moreover, it is event-triggered, meaning that immediately after the Provider Manager detects a dead provider, the Replication Module listens to signals and based on the information it holds, it is able to backup data. However, the module does not run on a genetic algorithm such as AGRA and does not move the replication scheme closer to the clients as ADR does. The main advantages the Replication Module brings to BlobSeer are: a reliable data storage service that can easily adapt to provider failures and a dynamically-adjusting replication scheme that is optimized to use less storage resources.

3. Background

3.1 BlobSeer

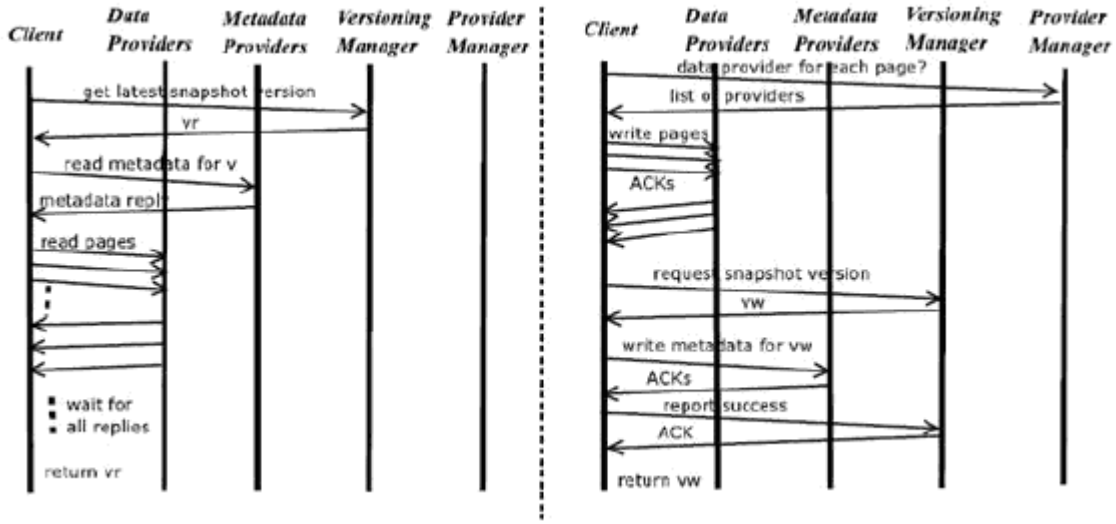
BlobSeer[1] is a data storage service designed to deal with requirements of large-scale data-intensive distributed applications. Data is abstracted as a large sequence of bytes and stored as a blob (binary large object), which is a very large file.
BlobSeer, among other data storage services, addresses storage challenges such as heavy access concurrency, blobs versioning, large aggregated storage space and fine grain access.
The fact that differentiates BlobSeer from the 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.

Fig. 1, BlobSeer Architecture



BlobSeer's current architecture [6] is illustrated in Figure 1. **Clients** initiate the read, write or append operations. There can be situations where many clients access the same resources for reading or writing. BlobSeer's support for concurrent operations [7] permits them fast and reliable access to the required resources. **Data providers** store the pages. They can automatically join or leave the system. The **providre manager** deals with provider monitoring and allocation strategy. **Metadata providers** store blobs' metadata. It enables access to blobs' metadata for offsets and different versions. The **version manager** holds the version of each file. It deals with the serialization of concurrent write / append requests and with the assignment of version numbers for each new write / append operation.

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



The interactions between BlobSeer's entities are illustrated in Figure 2.

Clients initiate the read, write and append actions [8].
A READ request in BlobSeer starts with the client optionally asking the Version Manager for the latest published version of a blob. If the specified version is available, then the client contacts the Metadata Providers to retrieve information for the requested blob. The client fetches the corresponding metadata and contacts the Providers in parallel to read and store the pages in a local buffer.
However, in the case of WRITE requests the client asks the Provider Manager for a list of providers that are able to store the pages of the new (or updated, in the case of append) blob. Then, it contacts the Providers in parallel and writes the pages to them. Each Provider sends back an acknowledgement to the client. When the client has received all the acknowledgments, it contacts the Version Manager to ask for a new version number. The version number is used by the client to generate the corresponding metadata. The new information must be updated and the client gets a version number for the update. The last steps are to add the new metadata in order to consolidate the new version and to report that the new version is ready for publication.
APPEND operations are only special cases of writes. Everything that is true for writes, is also true for appends, unless explicitly specified.

3.2 MonALISA

MonALISA[9] which stands for Monitoring Agents using a Large Integrated Services Architecture is able to provide complete monitoring for complex systems. The framework helps managing and optimizing the operation performance of Grids, networks and running applications in real-time. In our project, MonALISA plays an important role in computing metrics based on data collected from agents that monitor the providers and on the fluctuations of read/write frequencies for a file. The metrics will be used by the Adaptive Data Replication Module to 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 [10] .

The MonALISA Repository [11] provides the ability for clients to subscribe to a set of parameters or filter agents to receive information from all the services. This offers the possibility to present statistics such as global views. In our case, we have choosen the MonALISA Repository to monitor the data providers for parameters such as load, free memory and bandwidth usage.
The features that MonALISA provides makes it a good choice for monitoring a distributed storage system. It can monitor its predefined parameters as well as user defined ones. The monitoring of user defined parameters is possible due to an application instrumentation library called ApMON [12]. ApMON enables applications to send monitoring information to the MonALISA Service as UDP datagrams. Applications can report any type of information the user wants to monitor.

4. Replication Architecture Overview

The Adaptive Data Replication module architecture is illustrated in Figure 3. BlobSeer's current architecture consists of a provider manager, providers, metadata providers and a version manager. The Replication Module brings to BlobSeer two main elements: Listeners and the Replication Manager.

4.1 Listeners

The role of the Listeners is to send the required messages to the Replication Manager in order to assure that it has all the necessarily information it requires to work. Thus, Listeners send status update messages from the Provider Manager and the storage Providers. The status-update messages sent from the storage providers contain information about the availability of a provider, and the write-page events. A write-page event takes place when content is being written to a page on a blob and the Provider Manager allocates data providers to store the pages. As far as the status-update messages sent from the Provider Manager are concerned, they are triggered immediately after the Provider Manager acknowledges the unavailability of a storage provider and its contents need backup to preserve the replication scheme.

4.2 The Replication Manager

The role of the Replication Manager is to *maintain, increase and decrease* the replication factor for the blobs in the system based on received and monitored data.

4.2.1 Maintaining the replication factor

In order to maintain the replication factor for blobs, the Replication Manager receives two types of status-updates. The first one represents triggered status-update messages that are sent from the storage Providers and the Provider Manager. These messages inform the Replication Manager about the availability of storage providers. Each provider sends a status-update message to the Replication Manager when it becomes available on BlobSeer. Moreover, if it happens to become unavailable for any reason, the Provider Manager is the one that triggers the update to inform the Replication Manager about its unavailability. The second type of status-update messages are sent regularly from storage providers by the Listener mentioned above. They contain information about the recently written pages. These pages are stored by the Replication Manager on a hash-table that it holds for each provider. The fact that these messages are sent by the providers themselves and not by the Provider Manager gives many advantages. Among them we mention the reduced number of messages that are sent across the system and the increased accuracy for the pages that the the replication module maintains in the providers' hash-table and the actual information stored on the storage providers.
When a provider becomes unavailable, the Replication Manager knows what pages were stored on it and it can create copies to maintain the replication factor for the affected blobs. After the affected pages are copied, if that is possible among the available providers, the metadata information is updated.

Our Replication Module will communicate with the Provider Manager, the Metadata Providers and with the data Providers to perform the necessarily operations in order to maintain the replication factor. The Listener communicates with the storage Providers to keep up-to-date the Replication Manager with information regarding the providers' availability and to receive information regarding the page keys that have been written on them. Moreover, the Replication Module communicates with the providers when a dead provider is detected and its pages must be copied to others to maintain the replication factor for blobs. Finally, the Replication Module communicates with the Metadata Providers to update the metadata entries of each file that has been affected by the provider that went out of order.

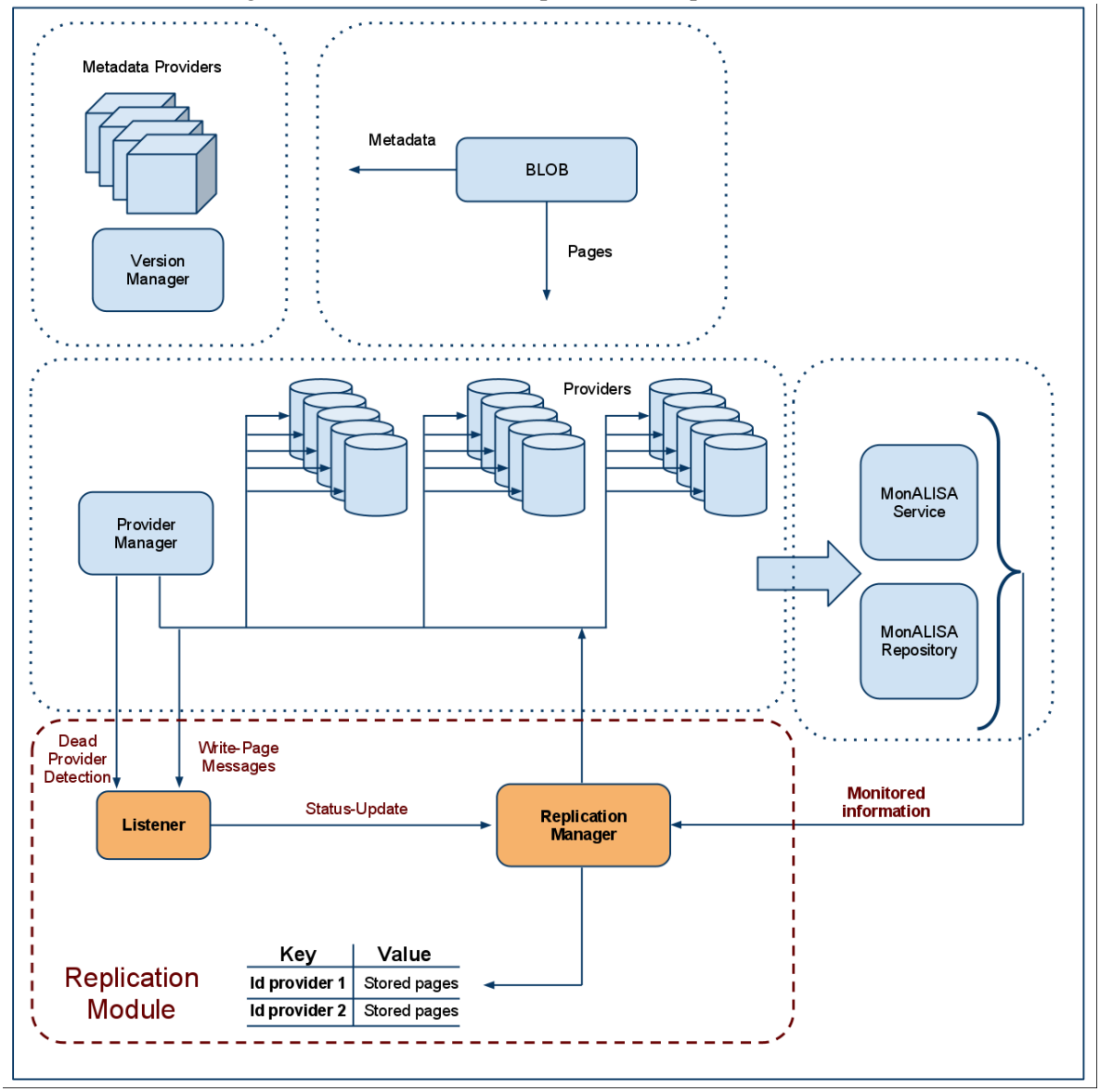
The main advantage of proceeding like above is the fact that everything happens in the background. The end-user continues working, even if a replica of one of the blobs he was reading from fails. There are other replicas on the system and the user can read them, without losing data.

4.2.2 Increasing and decreasing the replication factor

The second important role of the Replication Manager is the automatic increase and decrease of the replication factor. These changes occur as the number of blobs' reads vary. Moreover, the Replication Manager is able to prevent providers failure by monitoring the load on them. We have used MonALISA to monitor devices and access monitored data. There are two main sources for monitoring.
First, we have used the MonALISA Service to communicate with the monitoring support for BlobSeer. The monitoring support for BlobSeer records information such as blob ids, version, offsets, sizes, client ids, timestamps and stores them all on a database, thus making them accessible to the Replication Manager. The number of reads is used to compute a metric for each blob. The monitoring database is periodically queried to update the number of blobs' reads. If, for example the number of reads for a blob doubles or it drops below 10% from its maximum recorded value, the Replication Module decides whether to increase or decrease the blob's replication factor. The limits for increasing and decreasing the blobs' replication factor are easily adjustable. During this paper the limits are used for testing purposes. Thus, they are set to lower values than a real scenario would use.
Second, the load on providers is monitored using the MonALISA Repository that has to run on all providers. Currently we use Load5, free memory and the bandwidth usage to monitor all providers. When the load on a provider exceeds certain limits, the module automatically creates a backup of its pages in order to prevent data loss in case of provider failure.
The main advantage the two monitoring services bring is that the automatic change of the replication factor can be blob-based or page-based. For example, if certain pages on one provider are highly accessed and the storage provider becomes overloaded, the Replication Manager can increase the replication factor only for those pages stored on that provider. Otherwise when the number of reads for a blob increases, the Replication Manager can increase the blob's replication factor. Similarly, if the number of reads for a blob decreases and there seems to be no interest on its contents, the Replication Manager decreases, in time, its replication factor. More details about the limits and how the metrics are computed is given in the following chapters.

Below you can see BlobSeer's architecture with the Adaptive Data Replication Module. The figure shows the communication with MonALISA

Fig. 3 BlobSeer with the Adaptive Data Replication Module

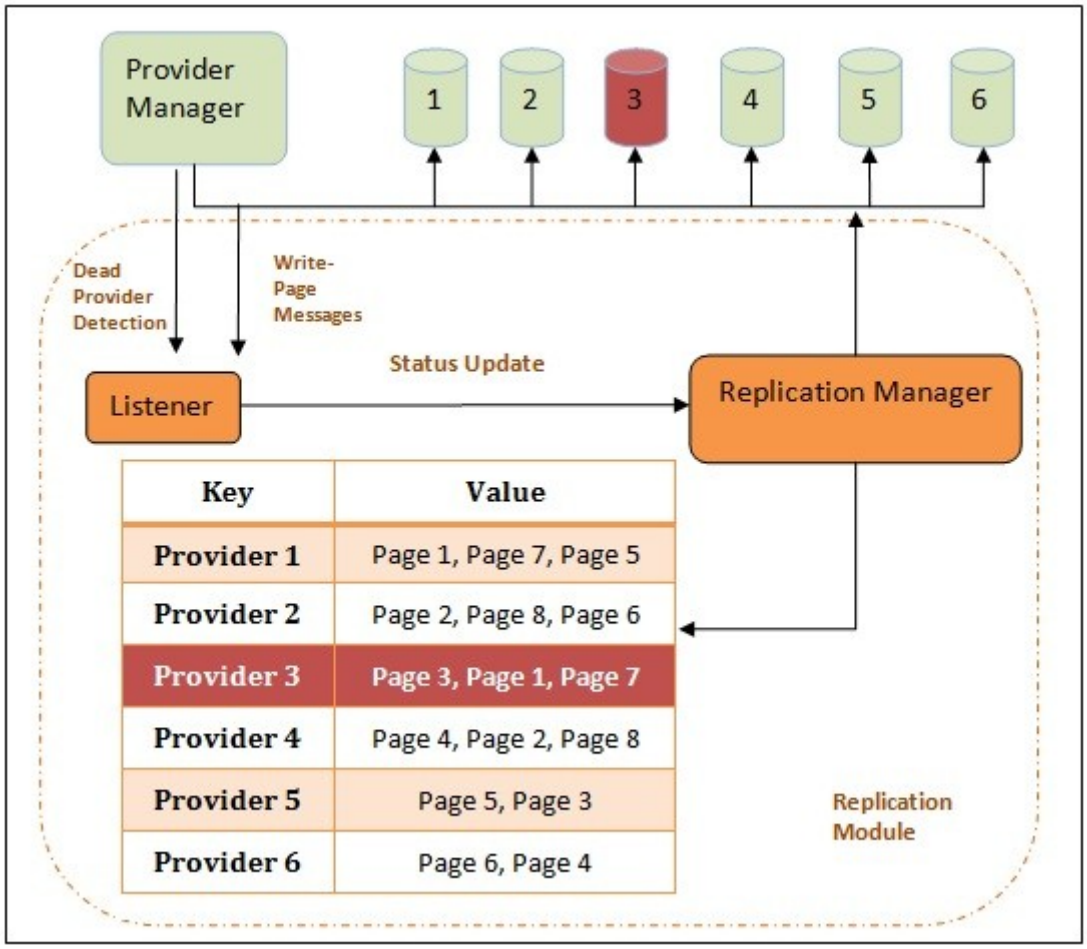


5. Illustration

5.1 Maintaining the replication factor

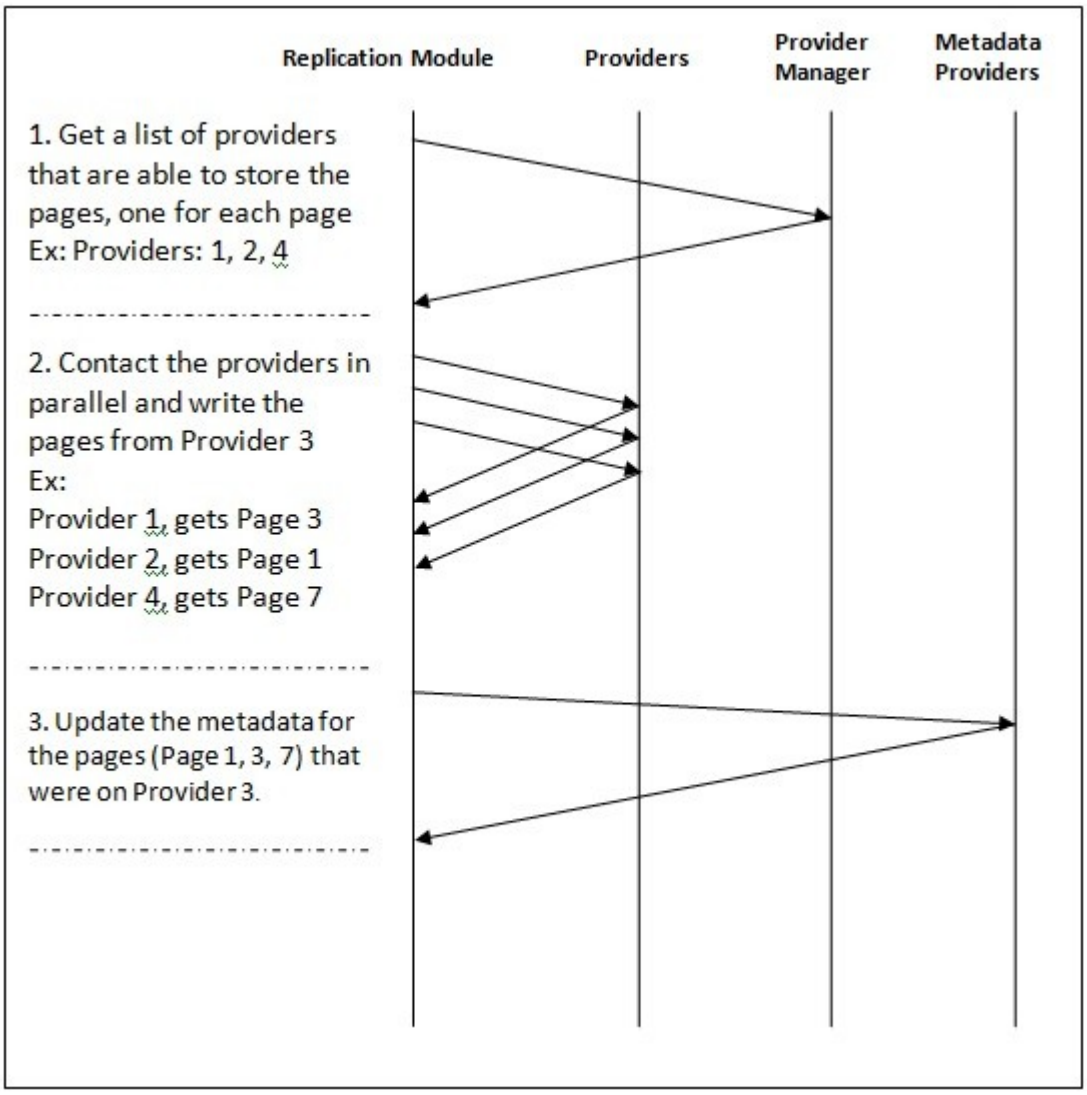
Figure 2 shows an example of how our module works to maintain the replication factor of a blob. We chose this scenario because it is a very common one. BlobSeer is a data storage service intended to be used within large scale distributed environments and offered as a service on clouds. It gives access to a high storage capacity. Therefore, it deals with a large amount of data storage providers to store the desired data. Because disks fail, BlobSeer's behavior is to see disk failure as something normal and not the exception. Our most important concern is to see what happens when a disk fails or there are communication problems with the data provider. In our example, we illustrate a topology with only 6 data providers, for simplicity reasons, where one of them fails. In this scenario, the provider's failure can be interpreted as disk failures, electricity issues or networking access problems.
In our example we consider that there is one blob of 8 pages with 2 replicas. When the blob is created, its data chunks are distributed among the 6 providers. We suppose that provider 3 fails to respond. Now, the Adaptive Data Replication Module has to copy each page stored on provider 3 to the working ones. It knows what pages it has to backup from the hash-table that saves the state of the providers. Thus, each data chunk from provider 3 will have a backup copy on another provider and the replication factor will be preserved. After the pages are copied, the corresponding metadata information is updated as well. Thus clients will refer to the new copies of the chunks and the old provider will not be accessed until it becomes ready.

Fig. 2, Scenario of the Replication Module, maintaining the replication factor



The following figure, Fig. 3, shows what actions are performed by the Replication Module for the above-mentioned scenario.

Fig. 3, Actions performed by the Replication Module
when a provider is out of order



In the above example, we assumed that there are no page duplicates on any provider. Page duplicates are not permitted because ultimately all blob's replicas could get on one provider making it a single point of failure. If we can't copy pages to a provider, then we add them to the unresolved list for further processing. Pages that are in the unresolved list have priority due to an aging mechanism when a new provider becomes available.

5.2 Increasing and decreasing the replication factor

The following 3 examples illustrate the Replication Module's interaction with the MonALISA Service and MonALISA Repository.

5.2.1 Increasing the replication factor

5.2.1.1 Using MonALISA Service

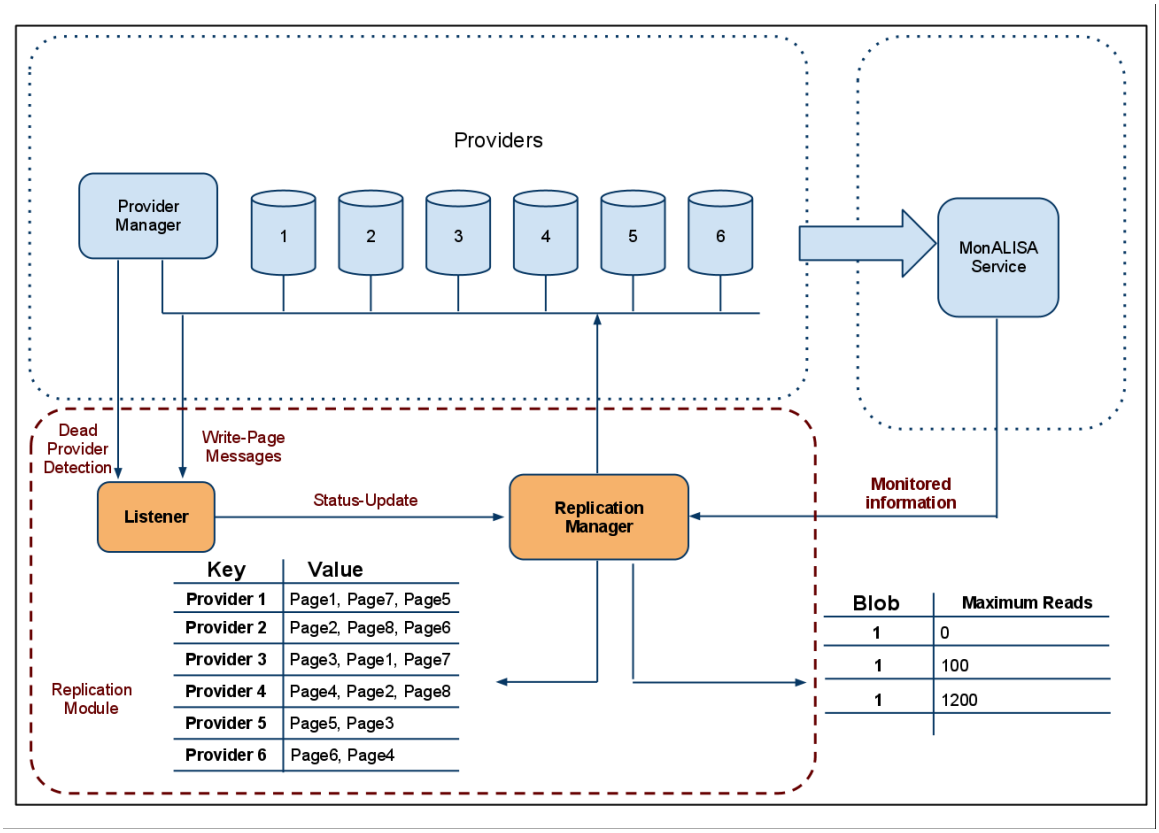
Figure 4 shows how the Replication Module collects data from the monitoring services for BlobSeer. It gets monitored data from MonALISA Service and stores it on special tables designed for BlobSeer's needs. The monitoring services collect useful information from BlobSeer, such as the write and read page events. For each of them it records a timestamp, the page's key and different

numeric values. All these are stored on separate tables for reads and for writes. The Replication Module queries periodically the monitoring services for BlobSeer at regular time intervals. We decided to use a time interval of 15 seconds. Each time it compares the results to the previous and based on statistics such as the maximum number of reads per time interval and the current one it makes decisions whether to increase the replication factor of that file. Currently BlobSeer increases the replication factor of a file if the current number of reads exceeds 2 times the maximum recorded number of reads. As it was mentioned above, these timers as well as all the parameters that we use can be easy customized.

First, after the blob was created and content was written to it, we read it 100 times in a time interval. On blob's creation, the maximum number of reads on a time interval will be reset to 0. Each read in a time interval will be counted and if it exceeds the maximum value by 2 times then it will become the maximum. In our case, the first maximum value will be set to 100. Since it increased from 0 - the implicit value at file's creation, the replication factor for the blob will not be updated. In another time interval, we decide to read the file 150 times. Because the Replication Module imposes a limit of 2 times the maximum recorded number of reads, although 150 is greater than 100, it will not replace that value. Thus, the replication factor for the file still stays the same.
Finally, we decide to read the file 1200 times. This time, 1200 reads exceed 2 times the maximum of 100 and the Replication Module makes the decision to increase the replication factor for the current file by 1. It also updates the maximum number of reads from 100 to 1200. Next time the replication factor for the blob will be increased is when the number of reads exceeds 2400 counts.

The time intervals to check the monitored values and the limit to update the maximum number of reads were designed to reduce the overhead this Replication Module brings to BlobSeer. As it was mentioned before, BlobSeer deals with large files and that is the reason why we should update the replication factor of a file only when it is necessarily, and not too often.

Fig. 4, Increasing the replication factor using
monitored information from MonALISA Service



5.2.1.2 Using MonALISA Repository

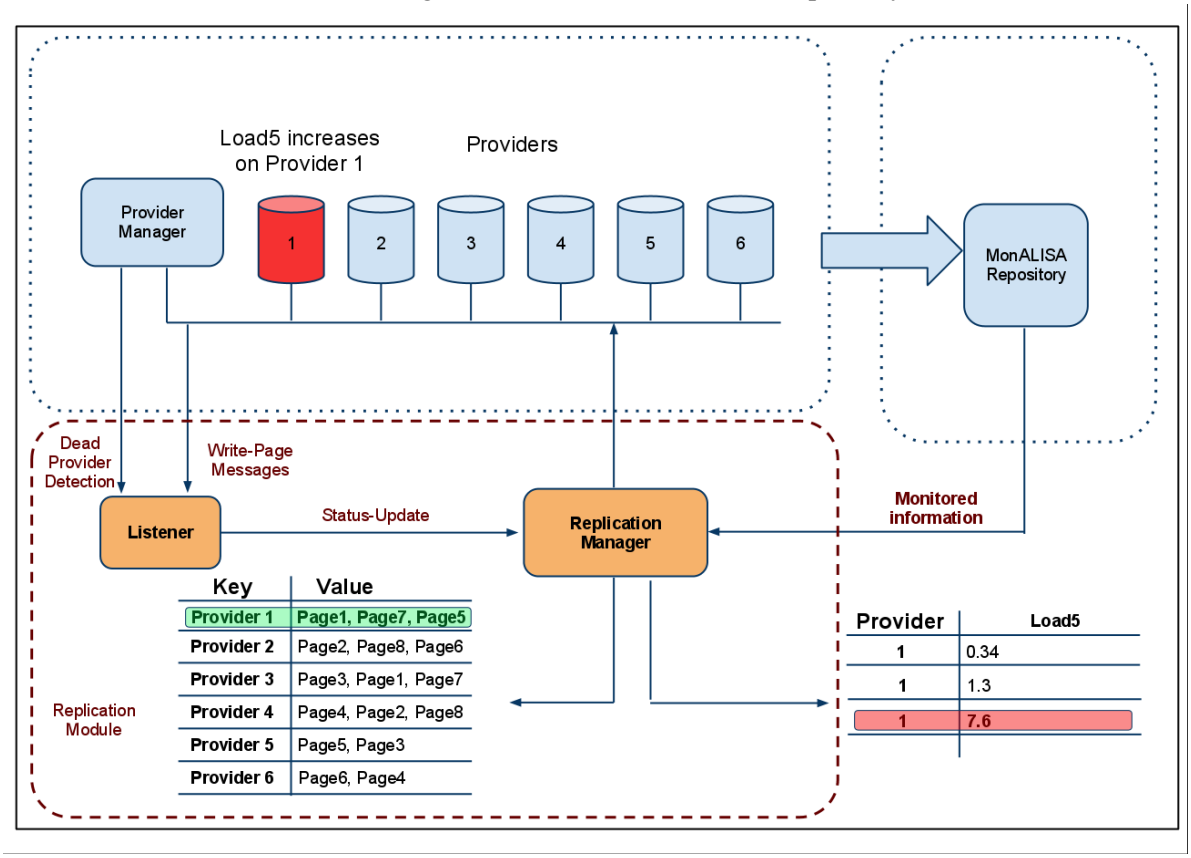
In Figure 5, we illustrate how we use the load monitored from the data providers to modify the replication factor of the blobs. We monitor each storage provider and store the values in MonALISA Repository. The current implementation evaluates the load on those machines. We query periodically the MonALISA Repository for information regarding the load on our storage providers. When we observe that the load increases above a limit, we decide that the provider should have a replica of its pages on a new one or on the others. An increased load on a provider could mean that it is more exposed to failure or service delays. The ability to create a provider's replica for the pages it stores brings the advantages of reducing the load on the machine and an available up-to-date copy of the provider in case it fails.

The illustration, presents a topology with 6 data providers, for simplicity reasons. The Adaptive Data Replication Module behaves the same whether it has to deal with a couple of data providers or with hundreds of them. For illustration, we consider that there is one blob of 8 pages with 2 replicas. When the blob is created, its data chunks are distributed among the 6 providers.

When the data providers start, the MonALISA Repository that runs on them monitors their load, free memory and bandwidth usage. The monitored parameters have equal proportions for computing the metric that is used to determine whether the provider is considered overloaded. In order for a provider to be considered overloaded, its load5 value should exceed 4, bandwidth usage should be more than

50% and the free memory should be less than 30%. If at least one of the conditions is met, then the Replication Module creates an exact copy of the pages the provider stores at that moment. This copy is distributed among the other providers unless they don't create duplicates and on new providers in the system if there are any.

Fig. 5, Increasing the replication factor
monitoring Load5 with the MonALISA Repository



5.2.2 Decreasing the replication factor

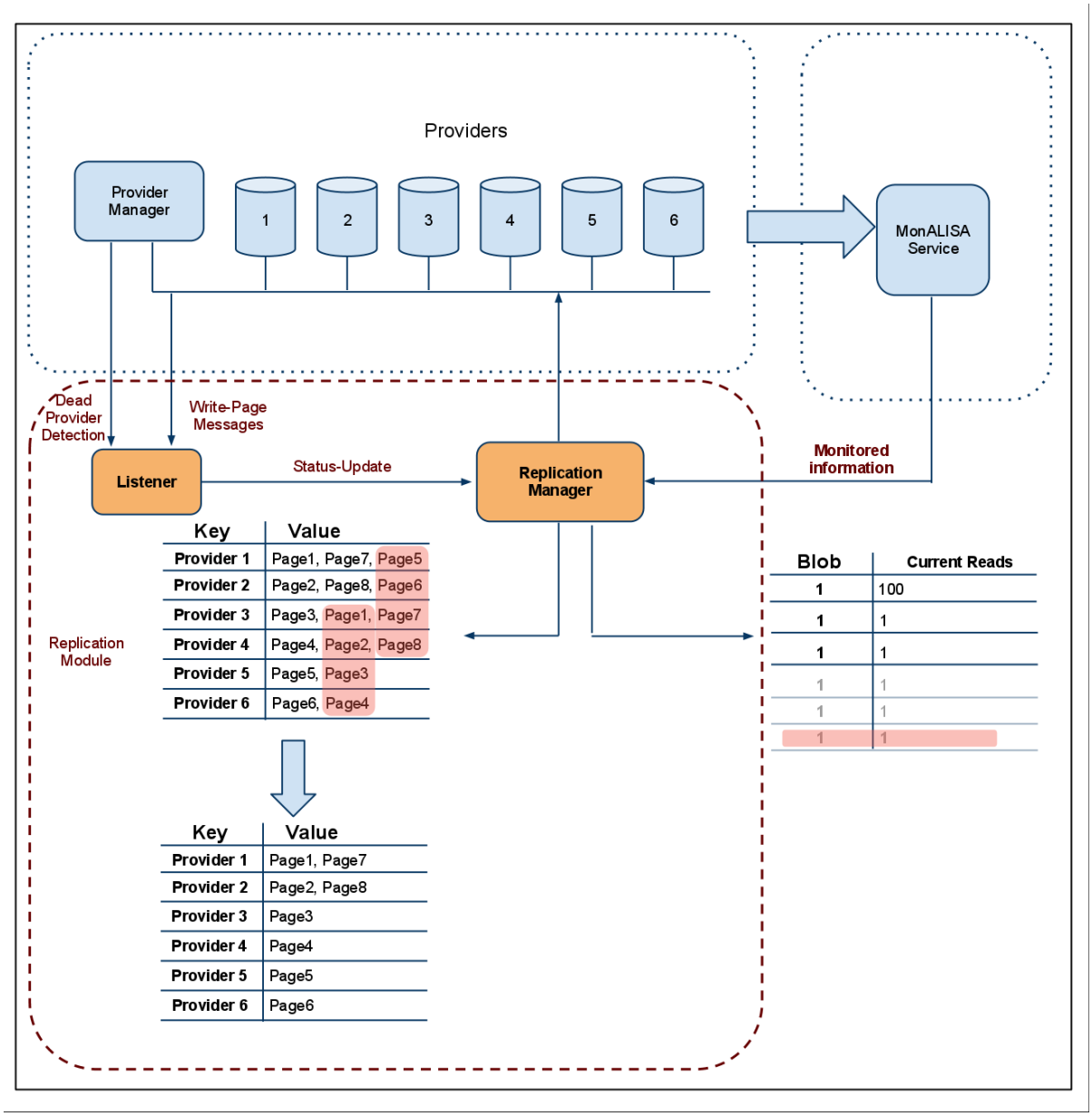
Figure 6 illustrates how the Replication Module makes the decisions when to decrease the replication factor for blobs. The action is based on the monitored information for BlobSeer from the MonALISA Service. The evolution of read patterns plays an important role in deciding when to increase or decrease of the replication factor. Thus, each 8 hours, the time interval to check monitored data, our Replication Module will query the databases to check for new udpates. It counts the number of reads per blob for the current time interval. If the number is below 10% the maximum recorded number of reads, it increases a below-limit hit counter. This hit counter indicates how many time intervals the number of reads has decreased to less than 10% of the maximum reads. If the hit counter reaches the value of 6, that means at least 48 hours the number of reads for a blob is below 10% of the maximum one, then we can decrease by 1 the replication factor of that blob. This brings us the advantage of freeing used space that automatically reduces costs for cloud implementations.

In our example, we illustrate a topology with 6 data providers, for simplicity reasons. The Adaptive Data Replication Module behaves the same whether it has to deal with a couple of data providers or with hundreds of them. For illustration, we consider that there is one blob of 8 pages with 2 replicas. When the blob is created, its data chunks are distributed among the 6 providers.

After the blob was created and content was written to it, we read it 100 times in a time interval. On blob's creation, the maximum number of reads on a time interval will be reset to 0. Each read in a time interval will be counted and if it exceeds the maximum value by 10 times then it will become the maximum. In our case, the first maximum value will be set to 100. Since it increased from 0 - the implicit value at file's creation, the replication factor for the blob will not be updated. Then for the next 5 time intervals, we issue only one read per time interval. After the sixth time interval (the first one read the file 100 times while the next 5, each one time), the Replication Module decides to reduce the replication factor of the blob.
It is important to notice that the replication factor is reduced only by 1 at a time. Moreover, the replication factor won't be reduced if it is 1.

The time intervals to check the monitored values and the number of less-than 10% hits to be reached in order to decrease the replication factor were designed to reduce the overhead the Replication Module brings to BlobSeer. As it was mentioned before, BlobSeer deals with large files and that is the reason why we should decrease the replication factor of a file only when it is necessarily.

Fig.6, Decreasing the replication factor
using monitored data from MonALISA Service



6. Implementation Details

The Adaptive Data Replication Module is implemented as a separate module that communicates with the Provider Manager and with the Metadata Providers.
The project was divided in two main milestones. The first milestone consists of maintaining the replication factor for each blob that it stores and the second one integrates the MonALISA monitoring service.

6.1 Maintaining the replication factor

BlobSeer is now able to maintain the replication factor. To achieve this, we had to add a new module called the Replication Module. As mentioned above, this module consists of two components: the *Listeners* and the *Replication Manager*. The listeners work on the storage providers and on the Provider Manager and collect necessarily information for the Replication Manager. There are triggered and periodic updates. In the case of the triggered updates, the listeners on the storage providers inform the Replication Manager about the availability of a storage provider, while the listener on the Provider Manager informs it when a provider becomes unavailable. As far as the periodic updates are concerned, each provider sends the most recent written pages it stores to the replication module. All these messages that are exchanged are called status-update messages. Thus, the Provider Manager sends a status-update message informing the replication module about a dead provider while each provider will notify it about its availability and the pages it stores. All these messages that are exchanged between the listeners and the Replication Module consist of RPC calls.
The Replication Manager receives the status-update messages sent by the Listeners from the storage providers and from the Provider Manager. These messages help it on making decisions maintain the replication factor when it is informed that one provider became unavailable to the system. The Replication Manager has an updated view on the providers network and the pages they store. Thus, when one provider becomes unavailable, it knows immediately what pages were stored on it and starts creating backup copies. All this information is stored on a special hash-table. We called this hash table, *the providers map*. The providers' map keys include all the providers available on the system and their corresponding values, all the pages that are written on a provider. This hash-table is populated from the status-update messages that the Replication Manager receives from the Listeners. When a provider becomes available, a new key consisting of the provider itself is added to the hash-table and the list of written pages is null at first. Periodically, as pages are being written to the system, each provider sends to the Replication Manager the list of most recently written pages. These lists, from each provider are added to the corresponding keys in the hash-table. The time interval these lists are sent equals that used to announce the provider's availability to the Provider Manager and is 5 seconds (UPDATE_TIMEOUT). Every 5 seconds, the Replication Manager, as well as the Provider Manager get information from each provider.
The Replication Manager must be fast when it is informed that a provider became unavailable and a backup has to be created for its pages on new providers. That is the reason we choose to implement a hash-table because of the low search time. When we hear that one provider is unavailable we can find its pages in *O(1)*, that brings a plus to the overall performance of the Replication Module. This is also the case when we receive the periodic updates from the peroviders. Each status update with page keys is simply appended to the currently list of keys for the corresponding provider in minimum time.
Another data-structure that we implemented is a hash-table for the replication factor. This hash-table stores as keys all the page-keys on BlobSeer and as values, their replication factor. This brings great performance increases since there are many times when we must know how many replicas a page-key has or should have. Since the other method implied an RPC call and then a count of the number of

elements returned, our method provides a fast way to find the replication factor in O(1). Regarding the huge number of page keys that will be used on the system, this implies that a large amount of memory will be used to store this hash-table. Since in this case response time as well as processing time is critical, we choose to give time more weight than used memory. This hash table is populated each time status-update messages are received for the written pages by increasing the replication counter for the received key.

As it was seen in the illustration above, when one provider fails, the Provider Manager sends a triggered status-update message to the replication module to take action. The Replication Manager, once it receives the message, knows what pages have to copy from the providers map. It also knows how many replicas each page has in the system. The number of replicas is necessarily to count the number of pages that will be requested. We decided that for each page that for every page-key that needs backup to ask that key's replication factor pages from the Provider Manager. Thus, the Provider Manager simply adds the replication factors for the fallen keys and forms the required number of pages. It then asks the Provider Manager for the vector of providers that will be able to store them. The Provider Manager answers and now the page keys must be copied. Before copying a page to a provider in the list returned from the Provider Manager, we have to check for that page key's existence on that provider. If the page key has already one replica on a provider we cannot copy it again there. This will create duplicates on the system and is not permitted. Moreover allowing this would imply in time that all replicas of one page should get on a single provider making it a single point of failure for the availability of the entire blob. Thus, a page-key check is done and if it is found, the next provider in the list is being checked and so on. Now, there are two possibilities. The first one is to find a provider that has no replicas of the current page key and we can copy the page key to it without restrictions, thus maintaining the replication factor. But in the second case there is a chance that the list of providers the Provider Manager returns contains no provider that is able to store a replica of one page key. For example, when there are 4 providers, one of them fails and we had a blob with replication factor of 4. That implies that we are not allowed to create a backup for the failed provider on the remaining 3. Therefore, we decided to use another data structure, this time a vector of tuples. This vector contains each key that the Replication Manager was unable to store a copy when that was necessarily. We called this data structure the unresolved keys list. An example of how the unresolved keys list looks like:

```
( <page key1, unresolved replicas1>, <page key2,unresolved replicas2>, ... )
```

The tuple (in the example is marked between < ... >) consists of the page key and the number of unresolved / failed replicas for it. Because the unresolved keys list contains pages that were unable to be copied on the remaining providers, it will be processed only when new providers are available on the system. Thus, each time a new provider becomes available and sends the triggered message to the replication module, the Replication Manager starts processing the unresolved keys list. The list of unresolved keys has priority when a new provider is available. It contains sensitive data that is critical to be processed. Immediately when the Replication Manager hears of the new provider, it checks if there are any page keys in the unresolved keys list. If there are, it begins a similar process to the one described when a provider fails. This time we know for sure that as long as the new provider has free space, each page on the unresolved keys list can be copied on it as long as it creates no duplicates. As it was mentioned above, the list of unresolved keys holds a counter of unresolved replicas for every key. Therefore, we decided that when new providers become available to the system to decrease gradually the number of unresolved keys. That means creating a backup copy of each page in the list before moving to the second copy of a page key. This brings us the advantage that the overall "critical level" of the page keys in the list shows a smooth decrease. A page gets out of the unresolved keys list when the number of unresolved keys equals 0. When a provider is available and the list of unresolved keys is processed, there may be keys that will be removed and others that will remain in the list. The

keys that remained will have a priority to the new keys that will be added to the end of the list. Thus, when maintaining the replication factor and there is no possibility to create backup copies for page keys, we created a list of unresolved keys. This list has priority when new providers are available. Moreover, there is a priority mechanism for the elements in the list, the older they get, the first they will be processed.
For every page that needs a backup, the replication module first tries to copy it on new providers, then, if the copy was successful, it must update the metadata information. BlobSeer's metadata information is stored on the Metadata Providers. For fast access to it, metadata is organized as a tree with versioning information. Each blob has its own tree for every version. Tree nodes are distributed in a fine-grain manner namely, DHT. It gives full access to concurrent reads and writes. When the replication module updates the metadata information, it will use the two DHT methods, get and put respectively. Get will be used to retrieve information for a given key while Put is used to update the information for a key.

6.2 Integration with MonALISA monitoring service

The second milestone gives BlobSeer the ability to make decisions to automatically increase and decrease the replication factor for each blob or parts of it. These decisions are made based on monitored data that is checked periodically. In order to check periodically data for the monitoring services, we created two watchdogs that run as sleep for the specified time interval seconds and then query the databases. To achieve this milestone, the Replication Module had to firstly automatically increase and decrease the replication factor for blobs and secondly to be able to create a replica of all the pages that are stored on a provider. The following sections describe how we accomplished these two steps.

6.2.1 Increase and decrease the replication factor for a blob

In order to make a decision of whether to increase or to decrease the replication factor for a blob, we must store additional information for each blob on our system. Based on the additional information and the queries on the databases the Replication Manager makes the decisions to increase or to decrease the replication factor. In this situation when we query the database, we ask for information about the reads since we checked last time. The query returns with the pages that were written, the timestamp when it happened, the id, version and size of the blob. All these will be used by our data structures to make the information accessible to the Replication Manager. Thus, we have designed another hash-table that will index all the blobs in the system. Because we want to operate fast as well as not to waste too much memory where space can be saved, the keys of the hash-table are strings that concatenate "id version" of the blob. The id represents the blob's id and the version is exactly the random number that is generated when the page is written to BlobSeer. Each version of a blob corresponds to one random number that together with the blob's id, uniquely identifies a blob in our hash-table keys list. From now on, the version number will be identified as the random number generated by BlobSeer, unless otherwise specified. The blob's id and its version are returned by the query for each timestamp in the reply. So far, we have the key for our hash-table. We have designed the values in this hash-table to be tuples that contain the blob's number of pages, its page size, the maximum number of reads for it and the number of less-than 10% read hits.
The number of pages of a blob is computed when periodic write-page events are received from the data storage providers as described at the first milestone. The page size is returned by the query from the monitoring database.

With the information we already store, when we want to increase or to decrease the replication factor of a blob, we can simply generate its page keys that follow the pattern: <id, version, offset, page_size>. We have the number of pages and the page_size and with one iteration through the number of pages all the page keys are generated. This brings many advantages such as less time spent on finding the number of pages and the blob's page size since its id is in the hash-table's keys vector. Another advantage is that for every blob, there is only one record in our data hash-table and this saves memory.
For each time interval we want to find out what new changes occurred. All the monitored information is in the MonALISA Service's database. When the MonALisa Service starts, it loads several filters that wait for monitored data from BlobSeer to reach the service. When the filter receives data, it sends them to the other component, the monitoring database server, which stores them in special Postgres tables. Thus, the query will ask for all the reads that occurred since the previous call. For each line returned in the result, we counted the page reads of all blobs.

First we present a pseudo-code with the steps that our algorithm performs in order to increase the replication factor for blobs.

```
function increase replication factor
      query database → result

      for each element in result
            determine blob and increase its current number of reads

      for each blob that was read begin
            compare current read with Rmax
            if Rmax is 0
                  update first time the Rmax
                  continue with the next unprocessed blob
            else if 2 x Rmax < current reads then begin
                  - we have to increase the replication factor for the
                  current blob
                  - generate blob's page keys
                  if current page key does not exceed MAX_VALUE then
                        add each generated key to the unresolved keys list
            end
      end

      process the unresolved keys list

end function
```

In order to increase the replication factor of a blob, we have to determine the exact number it was read. We divide the number of counted reads to the number of blob's pages. The result is how many times a blob was read. This number is compared against the maximum number of reads, R_{max} . If the value of R_{max} is 0, then it means that we are checking for the first time the monitored information and we set it to the current number of reads. Otherwise we have to compare the number of reads against the maximum. If the current number of reads exceeds 2 times the maximum one, then we update the

maximum with it and start to generate the page keys. The page keys are generated as described above, with an iteration through the number of pages. Each generated page is pushed to the back of the unresolved keys vector unless it is not already there. If the page is already in the unresolved keys vector, then we update only the number of unresolved replicas for that page. After we push all the blob's pages to the unresolved keys, we call the function that deals with the unresolved keys vector and tries to copy those pages to new or existing providers without creating duplicates. If a new replica was successfully added, then we update the information for the number of replicas of that page. This information is stored on another hash-table as described at milestone 1 that indexes the serialized page keys and as values counts the number of replicas for a page. We decided to use the page keys as index and not the blob because this gives us the flexibility of modifying the individual replication factor for page keys and not for the entire blob.

Second, in order to decrease a blob's replication factor, the process is similar to the one mentioned above. For the beginning, here is the pseudocode for the function that deals with the replication factor decrease:

```
function decrease replication factor
      query database → result

      for each element in result
            determine blob and increase its current number of reads

      for each blob that was read begin
            compare current read with Rmax
            if Rmax is 0
                  update first time the Rmax
                  continue with the next unprocessed blob
            else if current reads < (1/10)*Rmax then begin
                  - we have to decrease the replication factor for the
                  current blob
                  - generate blob's page keys
                  for each generated page key do begin
                        if current page key's replication factor > 1 begin
                              get dht information for the current page key
                              erease one element from the dht
                              update the dht for the current page key
                        end
                  end
            end
      end

end function
```

The number of reads returned for the query is counted for each blob individually. The results are compared against the R_{max} value. If the current number of reads for a blob is less than 10% the R_{max} , then we increase the counter for less-than 10% hits in the touple for the blob. If the counter reaches 6,

then we add the blob to a vector whose elements represent blobs that will have the replication factor reduced by 1. It is important to notice that the replication factor for a blob will not decrease below 1 because that would mean to delete the file entirely and this is not permitted by BlobSeer's design rules.

When we designed the system, we planned to reuse the code. That is why, when we want to increase the replication factor for a blob we use the same vector as we would if we could not find a provider to store a key, namely the unresolved keys vector. The only difference is that in this particular situation, we do not wait for a new provider to become available, but we check immediately to copy the pages in it to the available providers. The function that performs the iteration in the unresolved keys vector, copies and updates the DHT information for the page keys is the same when increasing the replication factor for a blob with when a page could not be copied to the available providers and the Replication Module waits for a new data storage provider.

6.2.2 Create a provider replica

Load on a provider can increase to high values, and we might become concerned with its reliability. In this situation, a good approach would be to create a copy for all the pages of that provider. This will redistribute its load on the others. Our current data structures permit us to increase the replication factor not only for the entire blob, but for specific pages as well. As long as a page's replication factor does not exceed the MAX_VALUE, the Replication Module is allowed to increase it. The MAX_VALUE represents the upper limit for a page to increase its replication factor. The current implementation uses a value of 4 but that can be easily changed. MonALISA Repository monitors the load on providers. In our implementation, we have used only load5 to decide whether the provider needs a backup copy for its pages. When the load on a provider increases above 3, we start a process similar to the one described when the replication factor for a blob is increased. We generate each page key, we check if the page key has not reached yet its maximum replication factor and if another replica is permitted, we add the page to the unresolved keys list. If the same page was already there, then we increase the number of unresolved replicas for our page. Otherwise, the page is added to the back of the list for processing.
The upper limit of replicas for a page gives us more flexibility on dealing with the replication factor. Thus after creating a copy for all the pages on the provider, if its replication factor does not decrease, the whole process starts again. This repeats until either the page keys reach their maximum allowed replicas or the load decreases below the limit.

6.3 Issues

Work for the Replication Module was not an easy task. There were many challenges along the designing phase and the implementation. For the design phase, we first wanted the Provider Manager to be the only one who communicates with the Replication Module. After creating different scenarios, we found out that this design would increase the number of messages exchanged in BlobSeer for each write-page. So, we decided that the providers should communicate themselves the write-page events to the Replication Module. While this saved memory on the Provider Manager since each provider holds its own copy of the writen pages, the change only cut the number of exchanged messages by half because the Provider Manager was no longer involved in the write page events.

Another problem arose: how would the write-page events be transmitted to the Replication Manager? We have two solutions for it. Firstly, an intuitive answer would be to send them immediately as they occur. The advantage of this solution is that if a provider fails, the Replication Manager has up-to-date

information about the provider. More precisely, it has the most accurate information of the pages that were it stored. Secondly, each provider could build a local list of pages that it holds and send regularly the information to the Replication Module. This way, the Replication Manager may fail several copies if there were written pages since the last update until the provider failed. However, as easy and intuitive the first solution might sound, we decided to change it in favor for the second one. There was one big reason for that: the reduced number of messages exchanged with BlobSeer. Since the messages are exchanged periodically, the number of written pages can differ in size, but the number of messages stays the same. For testing purposes, we implemented the first solution as well, but found out that the Replication Manager would simply fail when a blob of 256 MB replicated 4 times was written on BlobSeer. The Replication Module got simply stuck with a lot of messages arriving at the same time. On the other hand, when we tested the second solution, everything went smoothly and the Replication Module correctly received information for the written pages from all providers.

In the implementation phase, we found out that informing the Replication Module from the Provider Manager when a provider becomes available was not a smart idea. To explain why, we have the following scenario: there are several providers and one of them fails. Suppose that we try to copy the pages it stored on the remaining ones but no other provider could accept pages since they would create duplicates. Thus, our design dictates that the pages are added to the unresolved keys list. We add those pages to the unresolved keys list and wait for a new provider to become available. Let's say that we are lucky and the new provider shows up and the Provider Manager just let us know about its availability. The same design tells us that the pages in the unresolved keys have processing priority and immediately after a new provider is available, we try to copy as many keys as possible from it. Said and done. The normal work-flow to write pages in BlobSeer is to firstly ask the Provider Manager a list of providers that are able to store them. Since our provider just became available in the system, its score is 0. Thus we make sure that the Provider Manager will include it in the list of providers. Theoretically all seems to be working just fine. The problem appeared when we tried to run it. It would wait and wait and eventually crack. After spending some time to analyse the problem we found that we created a deadlock by communicating with the provider manager. How did the deadlock appear? The answer is in the RPC messages exchange. The Provider Manager sent an 'up' message to the Replication Module informing it about the availability of the provider. The Replication Module, immediately checked the unresolved keys list and since it was not empty started to ask the Provider Manager if there is anyone who can store the pages the list contained. This is where the deadlock happened. Since we added the priority to the unresolved keys list we start processing it immediately a new provider becomes available. In our example, the Provider Manager waits for a confirmation message from the Replication Module that it acknowledged the availability of the new provider while the Replication Module asks the Provider Manager for a list of providers to store its pages. Both of them wait for each other to reply and no one can continue without an answer. That was the point where our system had to deal with a deadlock. We solved the issue by instructing the storage providers to send themselves that they are available to the Replication Manager.

The solution to the deadlock, eventually lead to another problem. This time, the problem was more subtle since it did not manifest at every run. There were cases when the Replication Manager was notified before the Provider Manager about the existence of the provider. If it happened that the Replication Manager wanted to copy pages on it, it was unable because the Provider Manager had no knowledge about the provider and could not modify its score. To solve the problem, we had to make several changes in the way the provider notifies the Replication Manager about its availability. It was clear that, by any means, the Provider Manager had to be the first to know about its existance and only

then the Replication Manager. Every provider communicates with the Provider Manager through periodic rpc calls. The time between two messages sent through rpc, is delimited by asynchronous waits of 5 seconds. Thus, the Provider Manager, receives every 5 seconds a presence update from every provider. In order to solve our problem, we moved the code that informs the Replication Manager about the existence of a provider between two async_waits. Thus, the Replication Module learns that the new provider is up 5 seconds later than the Provider Manager. This gives plenty of time to the Provider Manager to update the existence of the new provider and to make sure it is able to use it.

7. Evaluation

The Adaptive Data Replication Module that we added to BlobSeer deals with the problems of maintaining, increasing and decreasing the replication factor for blobs. As we already know, the Replication Module communicates with BlobSeer through RPC messages sent across the network. In order to evaluate our replication module, we plan to evaluate the overhead that it brings to BlobSeer. The overhead will be evaluated for two important factors: time and the number of exchanged messages.

7.1 Testing Scenario

All the tests presented below were done on a BlobSeer deployment of 4 providers. The 4 providers are runing on 2 workstations, namely 2 per workstation. In our tests we use equal-sized blobs of 1MB each with a replication factor of 2. Both workstations run on Dual Core processors with 2000Mhz processors. One of them, the remote one, has 2 GB of RAM while the other has 4GB. The two computers have each 5400 rpm hard-disks. The providers have a total space of 1GB. Each computer runs a 32bit Linux distribution. They are connected with a 100Mbps full-duplex link to a router. Each provider will be given 2GB storage.

7.2 Time evaluation

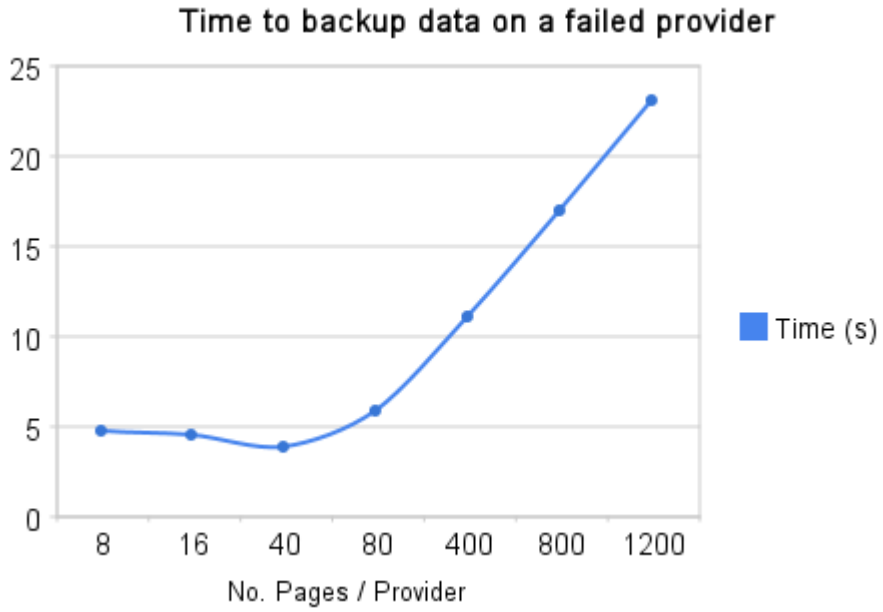
We intend to evaluate the time for maintaining the replication factor and the time for increasing and decreasing it.
First, we evaluate the time for maintaining the replication factor as the difference of time after a dead provider is marked as unavailable and the time the last page stored on it is copied to a new one or moved to the unresolved key. During the tests, we fluctuate the number of pages that are stored on the providers, thus increasing the workload on the Replication Manager.

7.2.1 Maintaining the replication factor

From the table below we can see that the time to create backup copies for the pages that are stored on a dead provider is increasing as the number of pages per provider increases. Although we increased the number of pages per provider, the increase ratio for the number of pages is less than the increase-ratio for the time.

No. Blobs	No. Pages / Provider	Duration
1	8	4.791574
2	16	4.518627
5	40	3.862959
10	80	5.921829
50	400	11.111309
100	800	16.974964
150	1200	23.124756

Below is the chart that illustrates our evaluation. With blue we have marked the number of pages that are stored on a provider. With orange, is represented the time it takes for the Replication Manager to move all the pages on a provider in case it fails. The effectiveness of our implementation consists in the reduced times compared to the large number of pages that have to be moved to the other providers.



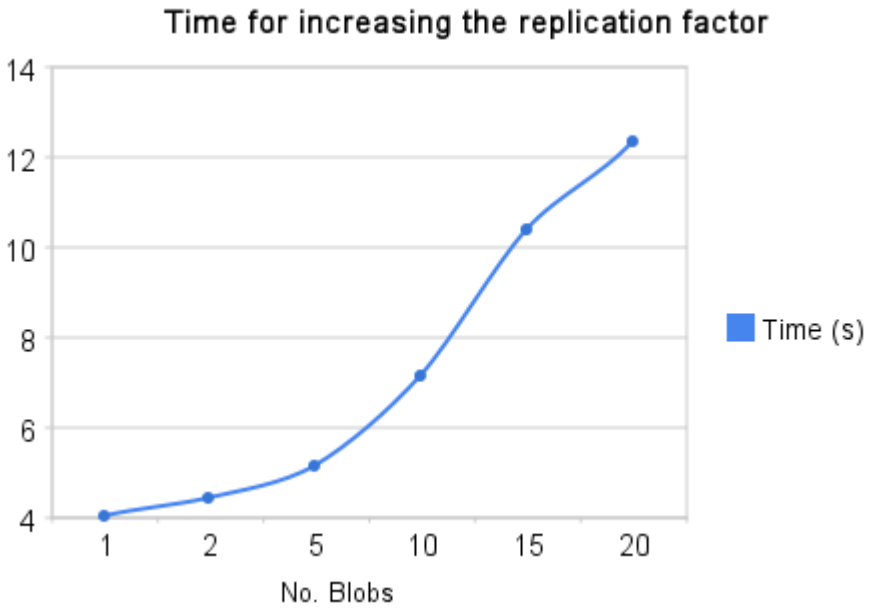
7.2.2 Increasing the replication factor

In order to increase the replication factor for a file, the conditions that were described in Chapters 6 and 4 must be met. The test was run each time the Replication Module queried the MonALISA

Service database for monitored information regarding the current READS. We measured the time since the Replication Module determined that the replication factor for a blob should be increased, until the last page for the last blob was written. The higher the number blobs, the more the time it lasts to increase the replication factor.

No. Blobs	Duration (s)
1	4.033224
2	4.429637
5	5.167387
10	7.145836
15	10.385917
20	12.372185

The chart below illustrates our results. This time, with blue we have represented the number of blobs and with orange, above them is the time it takes for the Replication Manager to increase the replication factor for them. For every test, the time illustrates the number of seconds that the Replication Manager works to increase the replication factor for all the blobs that are available.



7.3 Message exchange evaluation

The message exchange is the second overhead factor that we evaluated. Because BlobSeer is a distributed storage service, it relies on messages to communicate. Each message is sent using the

Remote Procedure Call (RPC) protocol using either TCP or UDP for the Transmission Layer. We intended to evaluate the number of messages exchanged by and from the Replication Module with BlobSeer. Because it is extremely large, we have created our own message monitoring watchdog that counts each RPC call, each database query and dht operations.

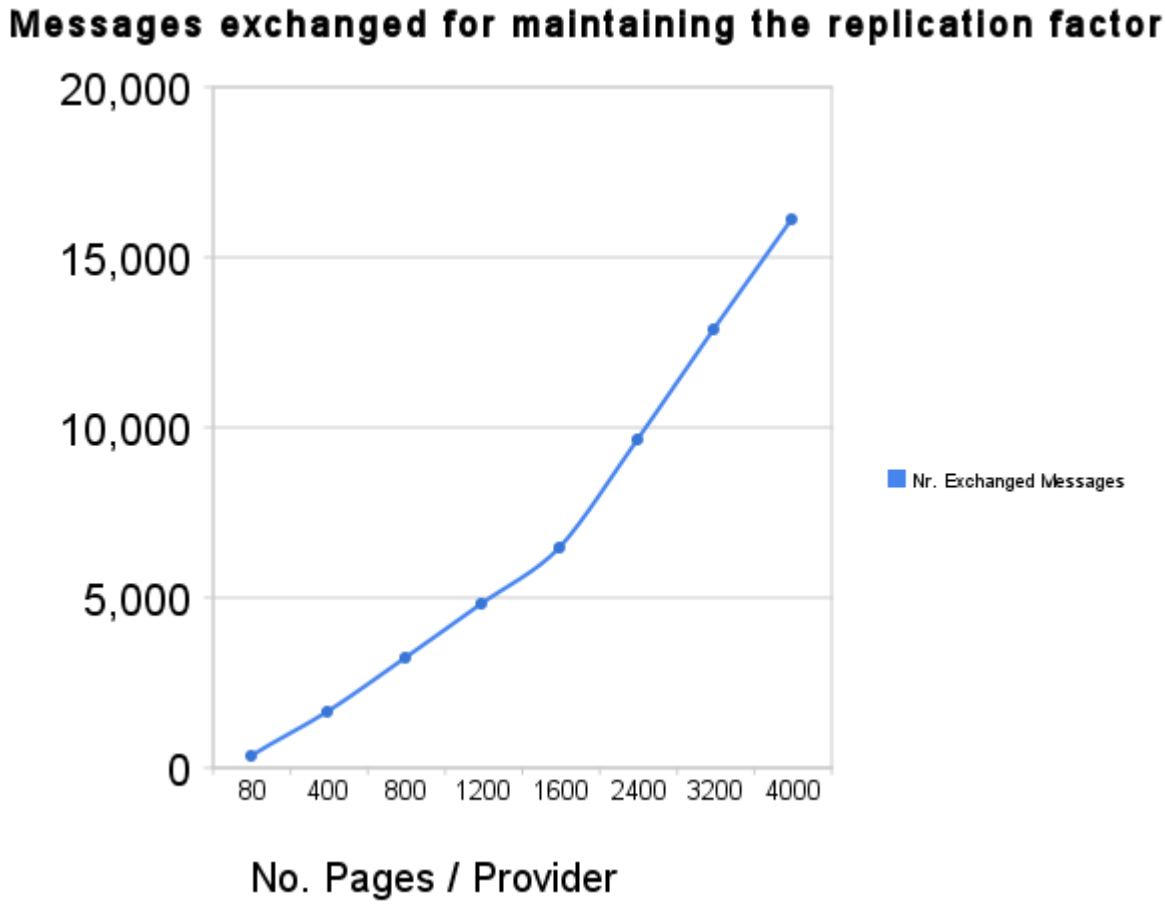
We have performed two category of tests. The first counts the number of messages that are exchanged in order to maintain the replication factor. The second counts the number of messages exchanged in order to decrease by 1 the replication factor for the given blobs.

7.3.1 Maintaining the replication factor

We have seen in section 7.2.1 how the time increases with the number of stored pages per provider. The situation stays the same here. As we increase the number of blobs, and the number of pages per provider as well, the exchange of messages rises. It is important to notice what the steps are for creating a copy for a blob in order to justify the values. The table below includes the details for the tests and the results.

No. Blobs	**No. Pages / Provider**	**No. Exchanged Messages**
10	80	335
50	400	1619
100	800	3226
150	1200	4831
200	1600	6442
300	2400	9666
400	3200	12894
500	4000	16118

The figure below contains the chart for the current test. With blue we have represented the number of pages per provider while with orange, the number of messages that were exchanged.

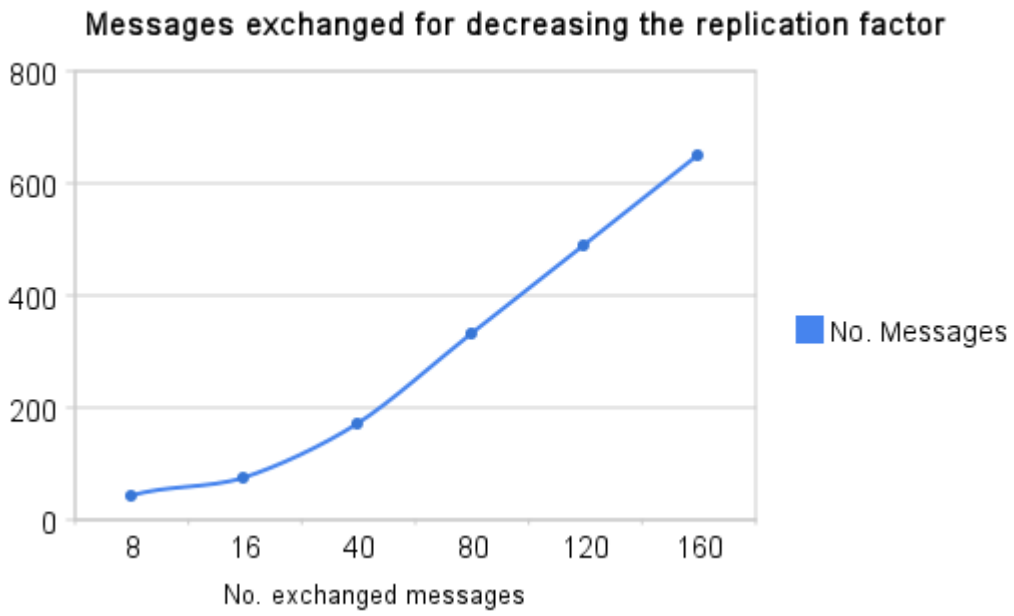


7.3.2 Decreasing the replication factor

So far, we tested the time and number of messages for maintaining and increasing the replication factor for blobs. In the current test, we watch the number of messages that our Replication Module exchanges with BlobSeer when it decides that the replication factor for blobs should be decreased by 1. The table below shows the number of messages that are exchanged in order to decrease by 1 the replication factor for all the blobs that are available. The number is compounded as the number of blobs times the number of pages times 2, for DHT GET and DHT PUT. In this situation, we have a constant increase ratio with the number of blobs, assuming each of them has equal page sizes.

No. Blobs	No. Pages / Provider	No. Messages
1	8	43
2	16	74
5	40	171
10	80	331
15	120	491
20	160	651

The chart illustrates the data from the above table. Still the number of messages (orange) is larger than the number of pages per provider (blue) but values are less than those for maintaining the replication factor.



7.4 Performance Evaluation

In order to evaluate the Replication Module's performance we take into account the convergence time. The convergence time represents the time that lasts from the detection of a dead provider until the replication factor of all the pages stored on that provider is reestablished. The convergence time is a sum of the time it takes for the Provider Manager to detect a dead provider (we call it T_d), the time necessarily to inform the Listener to send a status-update message to the Replication Module (we call it T_{su}) and the time it takes to get a provider for each page that was stored in the hash-table and to copy it (we call it T_{cp} per page). While the first timers until the data is copied are constant, the backup process varies in time depending on how many pages must be copied.

Therefore, the total convergence time will be: $T = T_d + T_{su} + n * T_{cp}$, where n is the number of data chunks that were stored on the dead provider and need to be copied.

The Replication Module performed well on the above tests. The results, are influenced by the time network delays and the number of packets loss. Since we used TCP for the Transport Layer, we experienced no loss of packets but this could represent a source for some delays. The Replication Module performed Well on all the tests. We checked the results after each test and BlobSeer was correctly recognizing a new replica or was able to perform READS and WRITES to new replicas or updated ones when providers were disconnected. There were several inconsistencies with the number of reads the replication module received from the MonALISA Service. The issues were due to the UDP messages the ApMon uses to transmit data. The problem was solved by increasing the UDP buffers for the MonALISA workstation.
Overall, the Replication Module achieved its goal on our tests, to maintain and to dynamically increase or decrease the replication factor of blobs based on monitored data.

8. Conclusions and Further Work

8.1 Contribution

In this paper, we have presented our approach towards creating an efficient adaptive data replication module for BlobSeer. The goals of the thesis were to implement a mechanism for BlobSeer that would enable it to maintain and automatically adjust blobs' replication factor. We have managed to satisfy all our initial goals and produce a stable and reliable Replication Module.
The integration with MonALISA monitoring service and MonALISA repository helped us in achieving our goal for accessing the monitored data. With the monitored data we were able to create metrics that were used to make decisions such as when the replication factor of a blob should be increased, decreased or even to detect a possible provider failure.

8.2 Future Work

Though the Replication Module is fully functional improvements can be made. A possible next step would be the optimization of the test results that were presented on chapter 7. As we have seen the duration and number of messages that was exchanged increased along with the number of blobs.
More advanced metrics could help the Replication Module to be more precise when modifying blobs' replication factor. It could also reduce the overhead that it brings to BlobSeer.
One of the improvements that the current system could benefit from would be a distribution of replication managers in order to eliminate the problem of single point of failure. Such a distribution would imply synchronization for the information that needs to be shared.

Acknowledgements

I would like to thank to Alexandru Costan, Alexandra Carpen and Bogdan Nicolae for their spport, feedback and ideas.

References

[1] 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”
[2] 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
[3] Ouri Wolfson, Sushil Jajodia, Yixiu Huang, “An adaptive data replication algorithm”, 1997, http://portal.acm.org/citation.cfm?id=249982
[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System", 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003. http://labs.google.com/papers/gfs.html
[5] Dhruba Borthakur, “HDFS Architecture” http://hadoop.apache.org/common/docs/current/hdfs_design.html
[6] 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
[7] Bogdan Nicolae, Gabriel Antoniu, Luc Bogue, “Enabling Lock-Free Concurrent Fine-Grain Access to Massive Distributed Data: Application to Supernovae Detection”, 2008
[8] Bogdan Nicolae, Gabriel Antoniu, Luc Bogue, “Distributed Management of Massive Data: an Efficient Fine-Grain Data Access Scheme”, 2008
[9] The MonALISA Project: http://monalisa.caltech.edu/monalisa__Publications__Papers.htm
[10] Mihaela Vlad, “Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System”, 2010
[11] MonALISA Repositories: http://monalisa.caltech.edu/monalisa__System_Design__repositories.html
[12] ApMon Library: http://monalisa.caltech.edu/monalisa__Documentation__ApMon_User_Guide.htm

Appendixes

A1 Communication with BlobSeer:

```
      boost::asio::io_service io_service;
      rpc_server<config::socket_namespace> rmanager_server(io_service);
      std::string config_file;
      config_file.append(argv[1]);
      repl_manager r_manager(config_file);

      //rmanager_server.start_listening(host, service);  does not work


      /*
       * registering the rpc and starting the service.
       */
      rmanager_server.register_rpc(RMANAGER_PROVIDER_DOWN,
                  (rpcserver_extcallback_t)
boost::bind(&repl_manager::provider_down,
                              boost::ref(r_manager), _1, _2, _3));

      rmanager_server.register_rpc(RMANAGER_STATUS_UPDATE,
                  (rpcserver_extcallback_t)
boost::bind(&repl_manager::status_update,
                              boost::ref(r_manager), _1, _2, _3));

      rmanager_server.start_listening(config::socket_namespace::endpoint(
                  config::socket_namespace::v4(), atoi(service.c_str())));

      INFO("listening on " << rmanager_server.pretty_format_str());
      io_service.run();
```

A2 Actions performed when a provider becomes unavailable:

```
INFO("Provider: [" << provider << "] is: " << message);

/*
 * If provider is down
 */
if (message.compare("unavailable") == 0) {

      //if it holds no keys - exit.
      if (providers_map[provider].size() == 0)
            return rpcstatus::ok;

      metadata::replica_list_t adv;
      metadata::query_t crt_page_key;
      unsigned int no_pages_processed;
      std::string adv_provider;
      std::stringstream page_key_s;

      /*
       * We have to compute the number of required pages from the PM
       * This number is the sum of the replica_counts for all the
                  pages
       * stored on the unavailable provider.
       */

      uint64_t num_pages = 0;

      DBG("we have to move the following pages:");

      for (i = 0; i < providers_map[provider].size(); i++) {
            page_key_s.str("");
            page_key_s << providers_map[provider][i];
            DBG("[" << provider << "] "
                        << providers_map[provider][i]
                        << " X "
                        << key_replcount[page_key_s.str()] );

            num_pages += key_replcount[page_key_s.str()];

            /*
             * Important: downgrade the replication factor for the
                        key
             * It will be updated when the key is copied
             */
            key_replcount[page_key_s.str()] --;
      }

      DBG("We require " << num_pages << " pages from the Provider
                                                      Manager");

      res = get_new_providers(adv, num_pages);

      if (!res || adv.size() < providers_map[provider].size())
            ERROR("could not get the list of providers");
      else
```

```
      DBG("got the list of new providers");


/*
 * parse the list of adv providers and try to get one for each
     page.
 * One provider is eligible to store a copy of a key if it does
     not
 * already have a copy of that key.
 */
for (no_pages_processed = 0; no_pages_processed
            < providers_map[provider].size(); no_pages_processed++)
      {
      has_provider = false;
      crt_page_key = providers_map[provider][no_pages_processed];

      for (i = 0; i < adv.size(); i++) {
            /*
             * if the adv provider is not already used
             */

            if (adv[i].host.empty() && adv[i].service.empty())
                  continue;

            adv_provider.clear();
            adv_provider.append(adv[i].host);
            adv_provider.append(":");
            adv_provider.append(adv[i].service);
            adv_provider.append("\0");

            /*
             * check if the current adv provider holds a copy
                 of the current key
             */
            for (j = 0; j < providers_map[adv_provider].size();
                                                    j++)
                  if (providers_map[adv_provider][j] ==
                                                crt_page_key)
                        break;

            /*
             * if it does, try the next adv provider.
             */
            if (j < providers_map[adv_provider].size())
                  continue;

            /*
             * otherwise, copy the current page to the new adv
                 provider.
             */
            DBG("Page [" << crt_page_key
                        << "] will be copied to new provider:
                        ["
                        << adv_provider << "]");

            res_change = change_replica(crt_page_key, host,
                        new_service,
                        adv[i].host, adv[i].service, false);
```

```
                /*
                 * mark the current adv provider as being used
                 */
                adv[i].host.clear();
                adv[i].service.clear();

                has_provider = true;
                break;
            }
            /*
             * if we could not get a provider for the current key:
             *  - we push it into the uresolved_keys list to try
                later when
             *  there will be many providers
             */
            if (!has_provider || !res_change) {
                INFO("Could not get a provider to store page_key:
                        ["
                        << crt_page_key
                        << "]. Page key added to the unresolved
                                                    list");

                //update the pages that will be required from the
                      Provider Manager
                page_key_s.str("");
                page_key_s << crt_page_key;

                //increase the number of failed replicas for the
                      current key.
                for (j = 0; j < unresolved_keys.size(); j++)
                      if (unresolved_keys[j].get<0> () ==
                                        crt_page_key)
                            break;

                /*
                 * if we found the key - increase the number of
                failed replicas
                 * otherwise add the new key with 1 failed replica.
                 */
                if (j < unresolved_keys.size()) {
                      get<1> (unresolved_keys[j])++;
                } else {
                      unresolved_t crt_unres;

                      get<0> (crt_unres) = crt_page_key;
                      get<1> (crt_unres) = 1; // we have one failed
                                                    replica
                      unresolved_keys.push_back(crt_unres);
                }

                //update the DHT for the current replica
                res_change = change_replica(crt_page_key, host,
                        new_service,
                        "n/a", "n/a", true);
            }
        }
```

```
/*
 * we have to remove the dead provider from the hash-table
 */
providers_map[provider].clear();
providers_map.erase(provider);
```

A3 Actions performed when a provider becomes available:

```
if (message.compare("available") == 0) {
            /*
             * We have a new provider in the system
             * We try to copy each page from the unresolved list to the new
                                                                    one.
             */
            providers_map[provider].clear();
            if (unresolved_keys.size() > 0)
                  check_keys_queue(new_host, new_service);
      }
```

A4 Status-update:

```
/*
 * The function deals with the status-update messages. These messages are
 * received periodically from providers when write-page events are
triggered
 * on them.
 * The function receives in params the page-keys written since the previous
 * call and adds them to the providers_map hash-table.
 */
rpcreturn_t repl_manager::status_update(const rpcvector_t &params,
            rpcvector_t &result, const std::string &sender) {


      bool local_result;
      DBG("GOT WRITE PAGE");
      metadata::query_t page_key;
      std::string service, provider;
      key_v received_pages;
      uint32_t i;
      std::string crt_id; //from the current page key: < id
                                                        the_random_value >
      std::stringstream aux_str;

      if (!(params[0].getValue(&service, true) && params[1].getValue(
                  &received_pages, true))) {
            ERROR("[" << sender << "] RPC error: at least one argument is
                                                                  wrong");
            return rpcstatus::earg;
      }

      status_update_counter ++;

      provider.clear();
      provider.append(sender.substr(0, sender.find_first_of(":")));
      provider.append(":");
      provider.append(service);
      provider.append("\0");

      DBG("status-update from [" << provider << "] got "
                  << received_pages.size()
                  << " page keys:");

      /*
       * add the received pages to the providers map
       */

      i = 0;
      while (i < received_pages.size()) {

            rpcvector_t v_params;
            metadata::root_t query_root(0, 0, 0, 0, 0);
            local_result = true;

            /*
```

```
             * store the replication factor for the current key
             */
            aux_str.str("");
            aux_str << received_pages[i];
            key_replcount[aux_str.str()]++;
            DBG("key: " << aux_str.str() << " key_replcount: "
                        << key_replcount[aux_str.str()]);

            /*
             * the key for the blob_aux_info map
             * string: (id version/rnd)
             */
            crt_id.clear();
            aux_str.str("");
            aux_str << received_pages[i].id << " " <<
                                              received_pages[i].version;
            crt_id = aux_str.str();

            /*
             * update the number of pages for the current blob/version
             */
            if (blob_aux_info.find(crt_id) == blob_aux_info.end()) {
                  get<0> (blob_aux_info[crt_id]) = 0; //nr_pages
                  get<1> (blob_aux_info[crt_id]) = received_pages[i].size;
                                                        //page_size
                  get<2> (blob_aux_info[crt_id]) = 0; //Rmax
                  get<3> (blob_aux_info[crt_id]) = 0; //nr_hits
            } else if (blob_aux_info[crt_id].get<0> () <
                                        received_pages[i].offset + 1) {
                  get<0> (blob_aux_info[crt_id]) = received_pages[i].offset
                                                             + 1;
                  DBG("crt_id: <" << crt_id << "> has: "
                              << blob_aux_info[crt_id].get<0>() << "
                                                               pages");
            }

            providers_map[provider].push_back(received_pages[i++]);
      }

      INFO("Providers_map[" << provider << "].size = "
                  << providers_map[provider].size());

      return rpcstatus::ok;
}
```

A5 Unresolved list checker

```
/*
 * check_keys_queue checks the unresolved list of keys and tries
 * to find a provider to maintain the replication factor for them
 */
void repl_manager::check_keys_queue(std::string new_host,
            std::string new_service) {

      uint64_t i, j, crt_key_no, required_pages = 0;
      metadata::replica_list_t adv, feasible_adv;
      bool res, res_add, no_change;
      metadata::query_t crt_page_key;
      std::string adv_provider;
      std::stringstream saux;
      p_map local_providers_map = providers_map;

      for (i = 0; i < unresolved_keys.size(); i++) {
            saux.str("");
            saux << unresolved_keys[i].get<0> ();
            required_pages += key_replcount[saux.str()];
      }

      DBG("We require " << required_pages << " pages from the Provider
            Manager");

      res = get_new_providers(adv, required_pages);

      if (!res || adv.size() < required_pages) {
            INFO("could not get the list of providers");
      } else
            INFO("got the list of new providers");

      /*
       * the list of feasible providers should contain only the new
      provider
       * The rest of providers cannot hold copies in order to avoid
      duplicates
       */

      feasible_adv.clear();
      if (new_host.compare("n/a") != 0 && new_service.compare("n/a") != 0)
      {
            for (i = 0; i < adv.size(); i++)
                  if (new_host.compare(adv[i].host) == 0 &&
                        new_service.compare(adv[i].service) == 0)
                        feasible_adv.push_back(adv[i]);
      } else {
            /*
             * TODO: check if no new provider
             */
            feasible_adv = adv;
      }

      /*DBG("List of feasible adv:");
      for (i = 0; i < feasible_adv.size(); i++)
```

```
      DBG(i << ". " << feasible_adv[i]);*/

adv = feasible_adv;
crt_key_no = 0;

while (1) {
      res_add = false;
      no_change = true;

      DBG("Processing "
                  << crt_key_no
                  << " key: "
                  << unresolved_keys[crt_key_no].get<0>());

      crt_page_key = unresolved_keys[crt_key_no].get<0> ();

      /*
       * find a provider and move the page's contents to it ->
            increase the
       * replication factor for the current page
       */
      for (i = 0; i < adv.size(); i++) {
            /*
             * check if the current adv provider already holds a copy
                  of
             * the current key
             */

            adv_provider.clear();
            adv_provider.append(adv[i].host);
            adv_provider.append(":");
            adv_provider.append(adv[i].service);
            adv_provider.append("\0");

            for (j = 0; j < local_providers_map[adv_provider].size();
                                                            j++)
                  if (local_providers_map[adv_provider][j] ==
                                                      crt_page_key)
                        break;

            /*
             * if it does, try the next adv provider.
             */
            if (j < local_providers_map[adv_provider].size())
                  continue;

            //DBG("adv[" << i << "]:" <<adv[i]);
            /*
             * otherwise, copy the current page to the new adv
                  provider.
             */
            INFO("Page ["
                        << crt_page_key
                        << "] will be copied to new provider: ["
                        << adv_provider
                        << "]");
```

```
                res_add = add_replica(crt_page_key, adv[i].host,
                                                    adv[i].service);

                if (res_add) {
                      no_change = false;
                      INFO("Added new replica of key:"
                                  << crt_page_key
                                  << ". Updating providers_map["
                                  << adv_provider
                                  << "]" );


      local_providers_map[adv_provider].push_back(crt_page_key);

                      /*
                       * decrease the number of unavailable replicas
                       * update the current adv provider as being used
                       */
                      adv.erase(adv.begin() + i);
                      if (unresolved_keys[crt_key_no].get<1> () > 1)
                            get<1> (unresolved_keys[crt_key_no])--;
                      else {
                            unresolved_keys.erase(unresolved_keys.begin()
                                        + crt_key_no);
                            crt_key_no--;
                      }
                      break;
                }

          }
          /*
           * if we could not get a provider for the current key:
           *  - we keep it into the uresolved_keys list to try later
           */
          if (!res_add) {
                INFO("Could not get a provider to store page_key: ["
                            << crt_page_key
                            << "]. Page key stays to the unresolved
                                                                list");
          }

          if (crt_key_no < unresolved_keys.size() - 1)
                crt_key_no++;
          else {
                crt_key_no = 0;
                if (no_change)
                      break;
          }

          if (unresolved_keys.empty())
                break;
      }

}
```

A6 Function to increase the replication factor:

```
/*
 * This function adds a new replica to a page.
 * The function performs the following 3 steps:
 *          1. Reads the page's content on a local buffer
 *          2. Writes on the new provider the contents of the page
 *          3. Updates the entry for the corresponding key in DHT
 *
 */
bool repl_manager::add_replica(metadata::query_t page_key,
            std::string new_host, std::string new_service) {

      //DBG("started add_replica");

      random_select vadv, vadv_copy;

      /*
       * Get page's replicas
       */
      //DBG("getting replicas for page_key: [" << page_key << "]");
      bool result = get_replicas(vadv, page_key);

      if (!result) {
            DBG("false result");
            return false;
      }

      vadv_copy = vadv;

      DBG("Page_Key: " << page_key << " has replica(s): ");

      int i = 1;
      while (1) {
            metadata::provider_desc adv = vadv_copy.try_next();
            if (adv.empty())
                  break;DBG("\n\t" << i << ". " << adv );
            i++;
      }

      metadata::replica_list_t adv_list;

      std::stringstream s1;
      s1.str("");
      s1 << page_key.id << " " << page_key.version;

      /*
       * Read page's contents and write data to the new provider
       * To read, we have to create a vector (page_result) that will have
       * a size = the number of pages current blob
       */
      boost::dynamic_bitset<> page_result( get<0>(blob_aux_info[s1.str()])
                                                            +1 );

      rpcvector_t write_params;
      write_params.push_back(buffer_wrapper(page_key, true));
```

```
buffer_wrapper page_contents;

vadv_copy = vadv;
/*
 * Fetching data to be read in a local buffer
 */
char *buffer = (char*) malloc(1 << 20 * sizeof(char));
result = read_page_content(page_key, buffer, "n/a", "n/a",
                                vadv_copy);

if (!result) {
      INFO("Could not read page " << page_key << " . Perhaps there "
                  << "are no more replicas");
      return false;
}

/*
 * Writing data from buffer to the new provider
 */

metadata::provider_desc adv;
adv.host = new_host;
adv.service = new_service;

write_params.push_back(buffer_wrapper(buffer, page_key.size, true));

direct_rpc->dispatch(new_host, new_service, PROVIDER_WRITE,
write_params, boost::bind(&repl_manager::rpc_write_callback, this,
boost::ref( page_result), boost::cref(adv), write_params[0],
write_params[1], page_key.offset, 0, _1, _2));

direct_rpc->run();

add_replica_counter ++;

if (!page_result[page_key.offset]) {
      ERROR("WRITE "
                  << page_key
                  << ": none of the replicas of page could be written
                        successfully");

      return false;
}

/*
 * Updating DHT entry with the new provider
 */
adv_list.empty();
vadv_copy = vadv;

i = 1;
while (1) {
      metadata::provider_desc adv = vadv_copy.try_next();
      if (adv.empty())
            break;

      adv_list.push_back(adv);
```

```
            DBG("\n\t" << i << ". " << adv);
            i++;
        }

        adv.host = new_host;
        adv.service = new_service;
        adv_list.push_back(adv);
        DBG("\n\tAdding to DHT:" << i << ". " << adv);
        i++;

        result = true;
        //DBG("PUT PAGE KEY: " << page_key);
        /*
         * Perform put to update the entry in DHT
         */
        dht->put(buffer_wrapper(page_key, true), buffer_wrapper(adv_list,
                    true), TTL, SECRET, bind(write_callback,
                                boost::ref(result), _1));
        dht->wait();

        dht_put_counter ++;

        if (!result)
            DBG("Something went wrong with put page key");

        INFO("Completed successfully.");

        return true;
}
```

A7 Function to decrease the replication factor:

```
/*
 * this functions reduces the replication factor for the given pages
 * by 1.
 * blobs are given by "id version" and all their pages are generated
 * inside.
 */

bool repl_manager::remove_one_replica(std::vector<std::string> pages) {

      metadata::query_t page_key;
      std::stringstream s1, s2;
      std::string blob; //only id and version
      uint32_t p_id, p_version;
      size_t pos;
      unsigned int i, x;
      bool result;

      for (x = 0; x < pages.size(); x++) {
            blob.assign(pages[x]);
            DBG("blob:" << blob);

            pos = blob.find_first_of(" ");
            s1 << blob.substr(0, pos);
            s1 >> p_id;

            s2 << blob.substr(pos + 1);
            s2 >> p_version;

            page_key.id = p_id;
            page_key.version = p_version;
            page_key.size = blob_aux_info[blob].get<1> ();

            for (unsigned int crt_page = 0; crt_page
                        < blob_aux_info[blob].get<0> (); crt_page++) {
                  page_key.offset = crt_page;

                  std::stringstream page_key_s;
                  page_key_s << page_key;

                  DBG("Page_Key: " << page_key << " X "
                              << key_replcount[page_key_s.str()]);

                  /*
                   * check the minimum permitted replication level
                   */
                  if (key_replcount[page_key_s.str()] <= REPL_LOWER_LIMIT)
                  {
                        DBG("Page key: " << page_key_s.str()
                                    << "Cannot decrease the replication
                                          factor below: "
                                    << REPL_LOWER_LIMIT);
                        continue;
                  }
```

```
//DBG("removing: " << page_key);

random_select vadv, vadv_copy;

/*
 * Get page's replicas
 */

result = get_replicas(vadv, page_key);

if (!result) {
      DBG("false result in get_replicas");
      return false;
}

vadv_copy = vadv;


/*
 * remove the first replica by try_next() and put back in
      dht the
 * remainder replicas
 */
metadata::provider_desc adv = vadv.try_next();
//DBG("Removing " << adv);
metadata::replica_list_t adv_list;

adv_list.empty();
vadv_copy = vadv; //the first element is skipped above in
                        adv.

i = 1;
while (1) {
      metadata::provider_desc adv = vadv_copy.try_next();
      if (adv.empty())
            break;

      adv_list.push_back(adv);
      //cout << "\t" << i << ". " << adv << endl;
      i++;
}

result = true;


/*
 * Perform put to update the entry in DHT
 */
dht->put(buffer_wrapper(page_key, true),
            buffer_wrapper(adv_list,true), TTL, SECRET,
            bind(write_callback,boost::ref(result), _1));
dht->wait();

dht_put_counter ++;

if (!result)
      DBG("Something went wrong with put page key");
```

```
                INFO("Successfully removed one replica for " <<
                                                        page_key);

                /*
                 * update the local replication counter;
                 */
                key_replcount[page_key_s.str()]--;

            }
        }

        return true;
}
```