Dynamic Provider Deployment using Monitoring Information

Alexandru Palade[1], Alexandru Costan[1], Valentin Cristea[1]

Alexandra Carpen-Amarie[2], Bogdan Nicolae[2], Gabriel Antoniu[3]

[1] Politehnica University of Bucharest, Romania

[2] University of Rennes 1, IRISA, Rennes, France

[3] INRIA, Centre Rennes – Bretagne Atlantique, IRISA, Rennes, France

alexandru.palade@loopback.ro, {alexandru.costan, valentin.cristea}@cs.pub.ro,

{alexandru.carpen-amarie, bogdan.nicolae, gabriel.antoniu}@inria.fr

May 2010

Abstract. In this paper we present an extension of the BlobSeer distributed storage system consisting of building the Dynamic Deployment Module (DD) which provides the ability of distributed storage systems to adapt to the users' necessities. We will describe how DD decides what is best for the entire system, especially how it takes the decision to expand or contract the pool of storage providers. We will present the system's architecture, the algorithms behind the hood and the intended testing framework.

Keywords: distributed storage, dynamic deployment, machine pool, data monitoring, scoring algorithm, blobseer

1 Introduction

1.1 Context

In today's world there is an increasing demand in storage space. Be it the need for scientific data, for complex applications or for anything else that need high storage capabilities, the need for files of size of Terrabytes magnitude is not a demand in the future anymore. However, there is also a need in redundancy and versioning of the data as safety is one of the primary concern nowadays. Moreover, in a distributed system there is also the problem of concurrency which has to be addressed in order to keep the data's integrity safe.

On the other hand, keeping the resources consumption at a minimum is a mandatory fact for every application which wants to be considered on a large-scale basis. The need for the resources to *scale up* is as important as the need to *scale down* together with the system overall load. Applications which are using the resources in a static manner are becoming obsolete because of their incapacity to adapt to the user needs. There is little sense in keeping thousands of machines started if only some percent of them are being actively used. The system has to scale down with its needs and to adapt quickly in order to conserve resources. This might not make sense at a first glance because we are used to Grid Computing, but consider the practical applications it can have in the context of *Cloud Computing* which is becoming less than a buzz word and more of a platform that everyone uses because of its flexibility and cost-reduction possibilities.

In this paper, we introduce the Dynamic Deployment concept, which enables a distributed storage system to scale up and down as it needs to. In the rest of the paper, we present our module(§1), how it integrates with the Blobseer platform(§2.1) and the monitoring module(§2.2), what algorithms it uses(§4.1) and examples of scenarios that might be applicable (§4.2, §4.3), what is the architecture(§5) and how we implement and test our use cases(§6 and §7). In the end we will also present some directions for future work(§8).

1.2 Motivation

Given the context, the possibility of adding and removing disk space on-the-fly is a feature that is not only good to have, but it is more of a *must have* feature due to the resources optimizations that are possible. As mentioned before, for any application running in a Cloud context, this module seems to be the optimal choice, as the service offers the infrastructure in a flexible manner.

1.3 Challenges

However, building such a module is not an easy task. There is an automatic decision that has to be made: how many resources do we need for the system to operate normally and at the same time to keep the resources utilization down to a minimum. This problem is addressed by using test-decided heuristic based on the monitoring data, which we will discuss more in the section Background Knowledge, Monitoring Data.

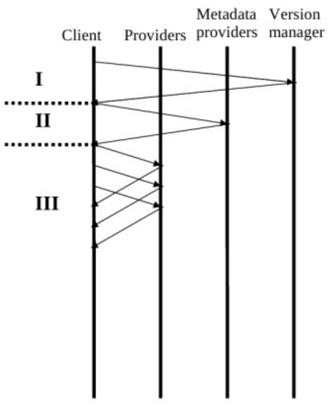
Another non-trivial problem that the module has to resolve is to make sure it keeps the data consistency, while adding or removing resources to and from the pool of the active ones. Take, for example, the removing of a resource from the system. This will have an immediate effect on the data that resides on this resource. Thus, the problem arises: what do we do with all the data? This problem is addressed later on in this paper, while we present Blobseer in the section Background Knowledge, BlobSeer.

2 Background Knowledge

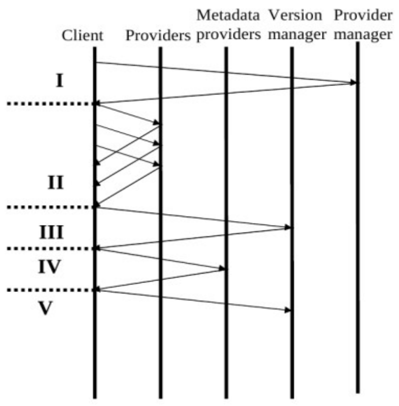
2.1 BlobSeer

BlobSeer[1] is a binary large object management service. It addresses the problem of storing and accessing very large, unstructured objects making use of a distributed environment. Each object is cut into *pages* and then distributed among different servers (Providers). The track of the pages is made by the metadata information which is kept on different servers (Metadata Providers). To avoid any concurrency issues, the data is versioned which also enables rolling back to different versions of the data.

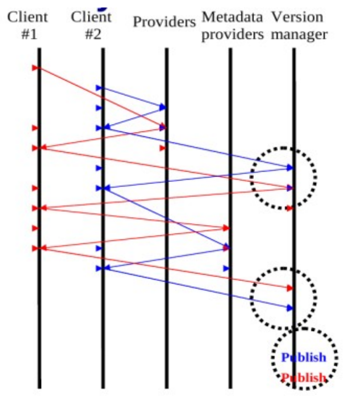
Except Providers and Metadata Providers there are also two other important actors in the system. The first one is Provider Manager which keeps information about the available data providers and schedules the pages' distribution among the providers. The second one is the Version Manager which is responsible for assigning version numbers to each of the client's requests. In the end, the Version Manager publishes the new version and makes it widely accessible.



Blobseer's READ operation



Blobseer's WRITE/APPEND operation



Blobseer – high concurrency scenario

The BlobSeer system is also capable to keep multiple copies of the same data (replication) in order to assure redundancy. This is a problem as the module has to make sure it keeps the replication factor *constant* while removing Providers from the active pool of available servers. Thus, the module will have to get the metadata informations for the data which lives on the Provider. Using this metadata, the module has to be capable to *move* the data from the current Provider to other active providers instead of just *discarding* it.

2.2 Monitoring data

Taking a decision which affects the entire distributed storage system can be considered to be critical, so a lot of attention must be paid to the data used in making such decision. The problem to be addressed when talking about the monitoring data is the place where we will take our data from. Because future heuristics to be developed might take into account both the physical factors of the provider and the Blobseer-related factors running on that provider, there is an obvious need to incorporate real-time data from both type of repositories.

To get the physical factors' values from all the Providers we will use MonALISA[2]. MonALISA stands for *Monitoring Agents using a Large Integrated Services Architecture* and has as goal to provide complete monitoring, control and global optimizations services for complex systems. Because of their highly-configurable system which can provide consistent data in real time, this system is perfect for our needs in gathering data about disk space, bandwidth usage, CPU load or any other physical factor that our heuristic might decide to use. The integration with MonALISA will be rather easy as its data can be exported using Web Services

To get the Blobseer-related factors' values we will use an already existing extension of the BlobSeer system[3]. Although the module's main usage is in User Accounting for a distributed storage system, the data collected by this module about Blobseer system turns out to be very useful to our goal. The underlying technology used by this module is the previously described MonALISA, with the advantage that it already aggregates the data and stores it in a persistent database, presenting us with interesting metrics like: number of reads / provider, number of writes / provider, etc. It is highly integrated with the Blobseer system, thus it can give us relevant real data about it.

3 Related Work

In the world of distributed systems there are a lot of solutions that try to resolve the problem of distributing large amounts of data over a grid. We can name a few examples like *GFS* - Google File System[4], *HFS* - Hadoop File System[5], Amazon's *S3* - Simple Storage System[6], DeepStore[7]. However, different solutions come with different features, as well as drawbacks. BlobSeer unique combinations of actors and the simple, yet efficient, architecture offers it a wide range of features.

	Fine Grain Access	Concurrent Reads	Concurrent Writes	Concurrent Appends	Versioning
Regular FS	X	-	-	-	-
GFS, HFS	X	X	-	X	-
S3	-	X	X	-	-
DeepStore	X	-	-	X	X
BlobSeer	X	X	X	X	X

Comparison table between existing file systems. [8]

Let us take the example of GFS. It is the technology that marks the basis of all other Google's technologies and products. Everything is built on top of GFS, thus inhereting the great power that comes with it, such as: concurrent reads, concurrent appends and fine grain access. This might be enough for Google's needs, but BlobSeer's architecture allows concurrent writes, a feature that few distributed storage system manage to achieve.

The module we are developing takes it a step further and brings a new set of features to the BlobSeer system. Currently, the BlobSeer storage nodes are statically specified in a configuration file. Hadoop File System does this as well, although you can add and remove data nodes on-the-fly. In HFS it is possible to exclude nodes one manually selects during run-time and the system takes care of replicating the data on other machines; however, the decision is not done automatically based on environmental factors. Also, in HFS it is possible to add nodes manually at run-time. The Hadoop system will redirect new write and read operations to these newly added blocks, although it will not rebalance the distribution of old data. This might be a good start, but there is still no way to do this in an automatically and optimal manner by a decision system inside HFS. An administrator has to scale up and down the number of available machines, in order to do what our module will achieve. Here, our module steps in, adding this important and non-trivial decision system. It will bring into BlobSeer the capability to dynamically adapt to the user's needs in order to keep performance top high, while cutting running costs down.

4 Scenario illustration

For an illustration on how the module will behave on certain patterns we will present an example of a possible scenario for the module's utility. Based on the monitoring data that we receive from the two repositories we are presented with the following values:

1. free disk space is above the 70% threshold
2. the replication degree is smaller than 3

3. we have a small read and write access rate to the data on this provider

Once the module reaches these conclusions, an action is triggered, specifically shutting down the provider, while making sure the data on the provider is moved and the replication degree is being kept. Shutting down the provider is simply a matter of calling an existing script; however, it is important for the module to fulfil the before-mentioned moving of the data and to keep the replication degree as we want to retain the consistency of the system's data.

Another example of a pattern in which an action should be triggered, might be:

1. read access rate per time unit is small
2. write access rate per time unit is small
3. the cost of moving the data is acceptable
4. there are providers that can take the load without a performance hit

Again, once the module finds this pattern, it decides to shut down the provider and make the necessary data moves.

4.1 Scoring Algorithm

The scoring algorithm, in its current form, provides a method to detect which Providers should be moved from the *Active Pool of Providers* (APP) to *Backup Pool of Providers*(BPP).

4.2 Factors

The factors to be taken into consideration can be divided into two subcategories:

1. *Physical factors*, depending on the physical machine that the module is currently analyzing (the machine that runs the Provider)
2. *BlobSeer factors*, metrics referring to Blobseer behaviour

Physical factors to be considered:

- Free Disk Space (f)
- Average bandwith usage (b)
- Uptime (u)

BlobSeer factors to be considered:

- Number of read accesses (r)
- Number of write accesses (w)
- Number of read accesses per time unit (rs)
- Number of write accesses per time unit (ws)
- Size of pages (p)

    - p = Page count * Size of page
- Replication degree (d)

4.3 Examples of scenarios

The decision which has to be made should be based on the following scenarios:

1. *f* >= 70%, *d* <= 3 and (*rs* has a small value or *ws* has a small value)
2. *rs* has a small value, *ws* has a small value and there are providers that can take the Provider's load

4.4 Heuristic

The heuristic which can present the weight factors are expressed in the following table:

No.	Factor	Weight factor / total score	Condition	Weight condition / factor
1	f	0.2	>= 95%	1
			>= 70% && < 95%	0.75
			>= 40% && < 70%	0.35
			>= 5% && < 40%	0.25
			< 5%	0.05
2	b	0.1	>= 60% (avg over 6h)	0
			>= 10 % && < 60% (avg over 6h)	0.5
			< 10% (avg over 6h)	1

4.5 The final score

Based on the factors described above, the final score for a provider will be computed after the formula

$$S=\sum_{i=1}^{n} wft_i \times wcf_i$$

where:

- wft_i represents the weight of the factor *i* from the total score
- wcf_i represents the weight of the true condition from the factor *i*

4.6 Decision

Based on the value of S (the final score) computed with the formula above, we can now take one of the following decision:

1. if $S < 0.4$, let the *Provider* in *APP*
2. else, move the *Provider* from *APP* to *BPP*

5 Dynamic Deployment Module

5.1 Architecture Overview

Although the problem solved by the DD module is complex, we tried to keep the architecture of the module simple. We divide the problem into two main parts: taking the decision and taking the corresponding action.

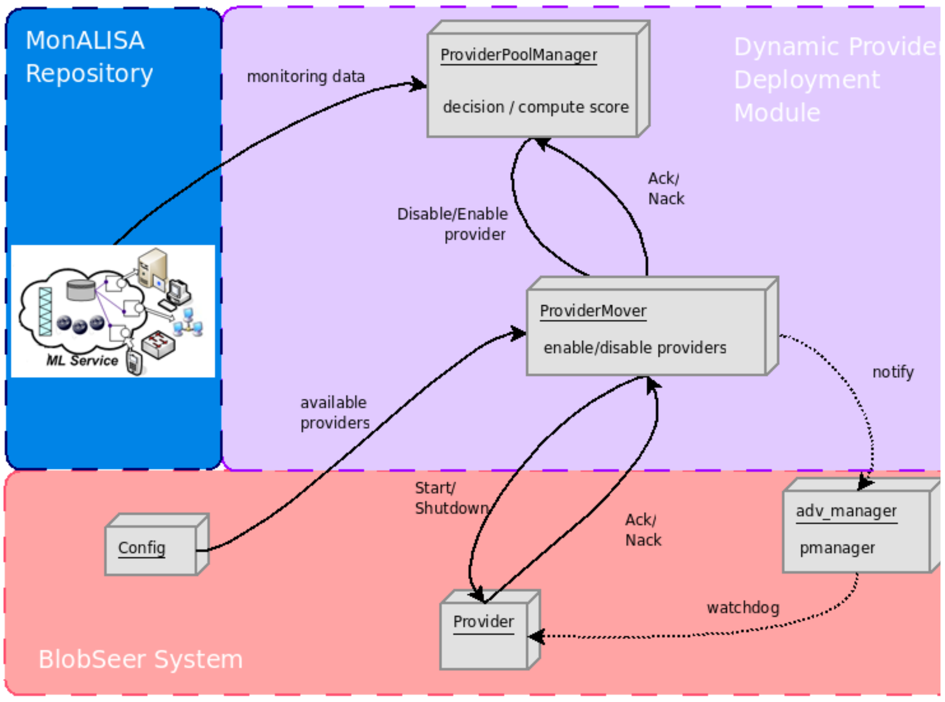
Taking the decision is the step responsible with retrieving the data and computing a score. The data is retrieved from two different sources, each one with different kind of metrics. To achieve this, we are using the monitoring module which is described in detail in the implementation section. Based on the data collected by the monitoring module, the decision algorithm can fire up and give a score as a result. The score is calculated based on the heuristics described in the scoring algorithm section. Once the score is obtained the module flow can jump to resolving the second problem.

In order to take the corresponding action based on the result obtained, the application needs to get a list of available servers (Data Providers) from the configuration module which can be turned on or off, depending on the decision taken. This part is also responsible for notifying the BlobSeer system, specifically the Provider Manager, of the changes made in the system. However, this step is not critical as the Provider Manager constantly verifies the Data Providers to see which ones are still alive.

5.2 Actors

The DD module interacts with the system using different components. They enable the module to communicate with different parts of the BlobSeer application, receiving and sending data as needed. The components (actors) of the module are:

1. *ProviderPoolManager* – class responsible to take the *decision*. The decision consists of enabling or disabling a number of Providers and it is taken on the basis of the information given to the class by the monitoring module.
2. *ProviderMover* – class responsible to make the movement of a Provider from *APP* to *BPP* or vice-versa depending on the commands that come from the *ProviderPoolManager*.



System Diagram for Dynamic Providers Deployment Module

Description

The DD module is responsible for two main actions:

1. establishing a score for each of the Providers
2. based on the score established at (1), to enable or disable Providers, moving the from APP to BPP or vice-versa

n order for the module to make action (1), it has to communicate with the Monitoring module and this is done through the ProviderPoolManager class. It will request monitoring data (CPU quota, HDD quota, load) based on which a *score* is going to be computed. Once the module has the score for each of the Data Providers it can now make a *decision* which Proviers are going to be moved from one pool of providers to another.

The decision is going to be put in practice by the ProviderMover class. It is the sole entity responsible for the two pools, *APP* and *BPP*, and the Providers' migration between the two. This is accomplished with the help of the following external components:

1. libConfig – it will communicate directly with this class from which it will get a list with available Providers and their address.
2. adv_manager – this class implements the Provider Manager. The ProviderMover will notify the Provider Manager of a change in the *APP*. In the case that this notifying fails, ProviderMover will not retry to do this as there is a watchdog facility already implemented in the system which scans the entire list of Providers and see who are alive or not.

3. Provider – the Provider class implements a Data Provider. The ProviderMover is going to communicate with it directly and issues commands like *start* or *shutdown* through which a Provider is moved from *BPP* to *APP* or, respectively, from *APP* to *BPP*.

6 Implementation

The architecture of the DD module is simple enough, however the implementation is not a trivial task.

The first complex problem is the framework for specifying the scenarios described in the scoring algorithm section. Because the list described in that section is not near to being exhaustive nor future-proof, the flexibility has to be the key component of this framework. Should this not be the case, the module will be rather useless as it cannot be expanded, as the number of use-cases for the module will increase. In its current form, this framework has the following flexibility points: specifying one or more factors for a specific scenario, each factor can have one or more conditions, the time interval for which a specific factor is to be considered. Also, the possibility to specify from which Providers to get data the data from is implemented. This being said, we consider that the framework in its actual form can accommodate the vast majority of the scenarios to be implemented.

Another complex issue to be solved is getting the monitoring data . Unfortunately, in this moment now, there is no common interface which can provide the application with all the data it needs in order to take the decisions it has to. For this reason, we are building this much-needed interface for all the read operations. This interface is responsible with retrieving data from both sources described in the monitoring data section. In the back it has to manage a couple of database connections, refresh them as needed and using them as optimal as possible.

6.1 Scenarios specifications

As described in Scenario Illustration section there are a lot of factors and, thus, patterns that might emerge which have to be taken into consideration. Because of this, there is a need for the specification framework to be flexible and easy-extensible. For these reasons, we have decided to take the most general approach we can for describing the scenarios. The user of this framework can specify the factor (including its weight in the total score), conditions to be true in order for the pattern to apply (including the weight of the condition in the factor's weight), the target (one, more or all providers) and the time interval. All of these are bundled into a scenario which ends up by giving a score. This score might be the most important value from the entire module and the way it is calculated once we have this framework in place is described in detail in the Scoring Algorithm section.

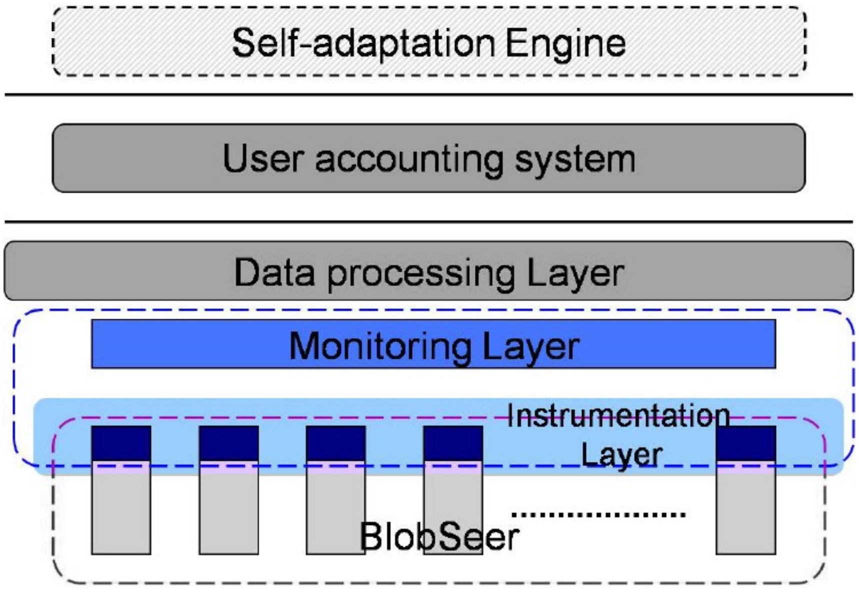
6.2 Monitoring Module

There are two main sources (named repositories from now on) from which the monitoring data will be fetched.

1. MonALISA repository
2. Internal Blobseer's monitoring database

Access to MonALISA repository will be made directly through the PostgreSQL[9] database system. MonALISA aggregates data from all nodes and sends them to a central repository where they are stored in a persistent database. The aggregation is done in different ways, depending on the configuration file of the repository. However, the table naming is consistent based on three different variables: *oldness* (how much to keep the data), aggregation factor and the factor's id. Based on the time interval on which we want to fetch the data, the monitoring module will have to be smart enough to take the decision from which table (specifically, which aggregation factor) to fetch data from. It is of no use to take data which has a 100 minutes aggregation factor when we are looking on a 10 minutes interval.

Access to the internal monitoring database is straightforward. The aggregation and manipulation of the statistic data is already done by another component[10] which is responsible for the user accounting of the BlobSeer system. This component will structure the backend MonALISA data in provider-oriented tables. Based on these tables we can access monitoring data like number of written pages, page size and timestamp.



User Accounting module architecture

7 Testing

We have established the module's architecture and the algorithm to compute the score based on the values received from the monitoring module. Next we plan to complete the implementation of this module and afterward to test it. Because there are heuristics on which the decisions are based, there will be some extensive testing to be done with different scenarios, factors and thresholds for the score. Based on this testing we will be able to provide a good default configuration of this module. Also, as a result of this test series, we can decide to add or remove some of the factors to be considered.

Experimenting will be conducted on the Grid'5000[1] testbed, a grid infrastructure distributed on 9 sites around France used for research in large-scale parallel and distributed systems. The general configuration for the nodes in the grid are Intel Xeon running at 2.3GHz, 8GB of RAM and a Gigabit Ethernet network.

8 Future work

Depending on the testing done, there is a lot more room to add new features and expand the module. The module is currently working towards integration with the BlobSeer system, however this should not be a requirement in the future. It can be expanded to allow it to be a plugable interface with minimum configuration requirements for easy integration and fast deployment with other distributed file systems. There is also room for improvement in the data movement operations after a new node is added or removed. In its current state this step is not optimized and for the future we are thinking about improving the system's behaviour when the pool of active nodes is modified.

References

1. V.-T. Tran, G. Antoniu, B. Nicolae, and L. Bougé, “Towards A Grid File System Based On A Large-Scale BLOB Management Service,” in Proc. EuroPar '09: CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service computing, Delft, The Netherlands, 2009: http://hal.archives-ouvertes.fr/inria-00425232/PDF/main.pdf
2. http://monalisa.cacr.caltech.edu
3. Mihaela-Camelia Vlad in the paper Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System
4. Google File System - http://labs.google.com/papers/gfs-sosp2003.pdf
5. Hadoop File System - http://hadoop.apache.org
6. Amazon's Simple Storage System - http://aws.amazon.com/s3/
7. DeepStore - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.5241&rep=rep1&type=pdf
8. Bogdan Nicolae, Luc Bouge, Gabriel Antoniu in the paper BlobSeer: dealing with increasing storage demands of large-scale data-intensive distributed applications
9. PostgreSQL relational database. http://www.postgresql.org/
10. Mihaela-Camelia Vlad in the paper Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System
11. The Grid'5000 project. http://www.grid5000.fr