Dynamic Provider Deployment using Monitoring Data

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

July 2010

Abstract. In this paper we present an extension for the BlobSeer distributed storage system which adds the Dynamic Deployment Module (DD). This module adapts the storage system to the environment by contracting and expanding the pool of storage providers based on the system's load. We will present the system's architecture and the reasoning behind it, the heuristic algorithm, the chosen implementation, and the experimental results obtained during testing.

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

Table of Contents

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 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. Cloud Computing is slowly taking over Grid Computing both in the industry and in the scientific world. The cloud is offering Internet-based computing which basically means that the power of a Grid is made accessible to anyone with a credit card. Therefore, we strongly believe that the BlobSeer system is well suited for a Cloud environment and, more specific, our module will reduce costs in operating BlobSeer on such systems.

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 the BlobSeer platform (§4), our module's architecture (§5) and the implementation, together with the algorithms and examples of scenarios that might be applicable (§6). Also we will present the reader with the experimental results (§7) and in the end with 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. Thus, we offer the possibility to take automatic decisions without human intervention bringing the power of self-adaptiveness to a distributed storage system.

We believe that such a module has uses in resource accounting in general, helping to evaluate how little or how much the resources are being actively used in a distributed system. This could also have a wide range of applications in job distribution, planning optimization or autonomic behavior.

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.

1.4 Terms

- **BLOB (Binary Large OBject)** – data structure used by **BlobSeer** to hold large (TB, PB) files. See section Background Knowledge, BlobSeer for more information about it.
- **Metadata Provider** – provider which holds **BLOB**s' meta-data. See section Background Knowledge, BlobSeer for more information about it.

- **Provider** or **Data Provider** – provider which holds the **BLOB**s stored by **BlobSeer** (only the raw data, no meta-data). See section Background Knowledge, BlobSeer for more information about it.
- **DD (Dynamic Provider Deployment)** – this is what we general refer to as the module that this paper describes.
- **APP (Active Pool of Providers)** – pool of Providers that are currently on and are actively used by the BlobSeer Infrastructure.
- **BPP (Backup Pool of Providers)** – pool of Providers that are currently off, waiting in stand-by to be activated in order to be used.

2 Overview

To make the Dynamic Deployment module work as needed, a handful of systems have to be integrated together. In the following sections we will present the main ones and how they interact with each other. At a glance, the module is attached as an extension to the BlobSeer system and feeds with monitoring data from a MonALISA repository. We have chosen to integrate informations offered by MonALISA because of its flexibility, persistent storage of monitoring data and high fault-tolerance.

2.1 BlobSeer Overview

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.

In order to work with large data objects, **BlobSeer** uses striping a process which will have as result the above mentioned pages. Pages will be then accessed by using a user-specified range in the form of (offset, size). The metadata which links physical pages to a snapshot's offset is organized as a

segment-tree like structure scattered among the system in a Distributed Hash Table[2]. This design enables direct and parallel I/O operations on the data while providing efficient fine-grained access *without* locking. Because data is an ever-changing resource, some kind of synchronization had to be made. In **BlobSeer**, this happens naturally by allowing BLOBs to have different versions as they are being added or updated by the user. Note that **BlobSeer** as a design-choice does not support deletion of BLOBs, just updates in order to provide a simple, yet powerful solution.

2.2 MonALISA and Monitoring Data Overview

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 both physical and internal factors' values for all the Data Providers in a system we will use the MonALISA[3] system. 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 in a direct manner. To get the BlobSeer-related factors' values we will use a MonALISA filter which enables us to fetch and aggregate all the data which is coming to a MonALISA repository. This filter is going to present us with interesting internal metrics about the BlobSeer System. Metrics including, but not limited to the number of reads / provider, the number of writes / provider, etc.

For the internal factors' values to arrive in a MonALISA repository there must be an **ApMon** client which enables the distribution of such metrics. Here we will be integrating an already existing extension of the BlobSeer system[4]. 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. It is highly integrated with the BlobSeer system, thus it can give us relevant real data about it.

2.3 Dynamic Deployment Module Overview

The module we are building is called **Dynamic Deployment Module** which integrates the before mentioned monitoring data, the BlobSeer distributed storage system and a heuristic-based decision-making algorithm. It enables BlobSeer in particular and distributed storage systems in general to dynamically modify its behavior as the need or lack of need dictates.

By developing such a module we are trying to address the problem described in the **Introduction** section of this paper. Because of the ever-increasing demand on storage space in the context of Cloud we will be able to cut costs while keeping the system running at its optimal level. This is done by continuously monitoring the system, applying tested heuristics and modifying the system state as needed. The heuristic used should enable the module to analyze and compute a score for any given Provider in real time for it to be useful. Although we will describe it in detail in a later chapter, for now this heuristic combines conditions and factors with different weights (importance) based on the values obtained from the monitoring data enabling it to provide a score for each of the Providers.

The Dynamic Deployment Module keeps interaction with the BlobSeer system at the minimum using it as rarely as possible. This was a design choice in order to keep it as loosely coupled as possible. In this way the module can be adapted easily in the future to work with other distributed storage systems given that we have a flow of monitoring data assured and we have a mean to easily start or stop machines that stack the data.

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[5], **HFS** - Hadoop File System[6], Amazon's **S3** - Simple Storage System[7], DeepStore[8]. 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.[9]

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 inheriting 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 give BlobSeer the capability to dynamically adapt to the user's needs in order to keep performance top high, while cutting running costs down.

4 BlobSeer - Data Storage System

BlobSeer is a system for managing huge amount of data in a distributed context. It handles large objects under the form of binary unstructured data providing the user with a transparent access to massive storage space. BlobSeer was designed for data-intensive applications which need fine-grain access to data. In order to enable a high performance for this kind of system in a heavy-concurrency

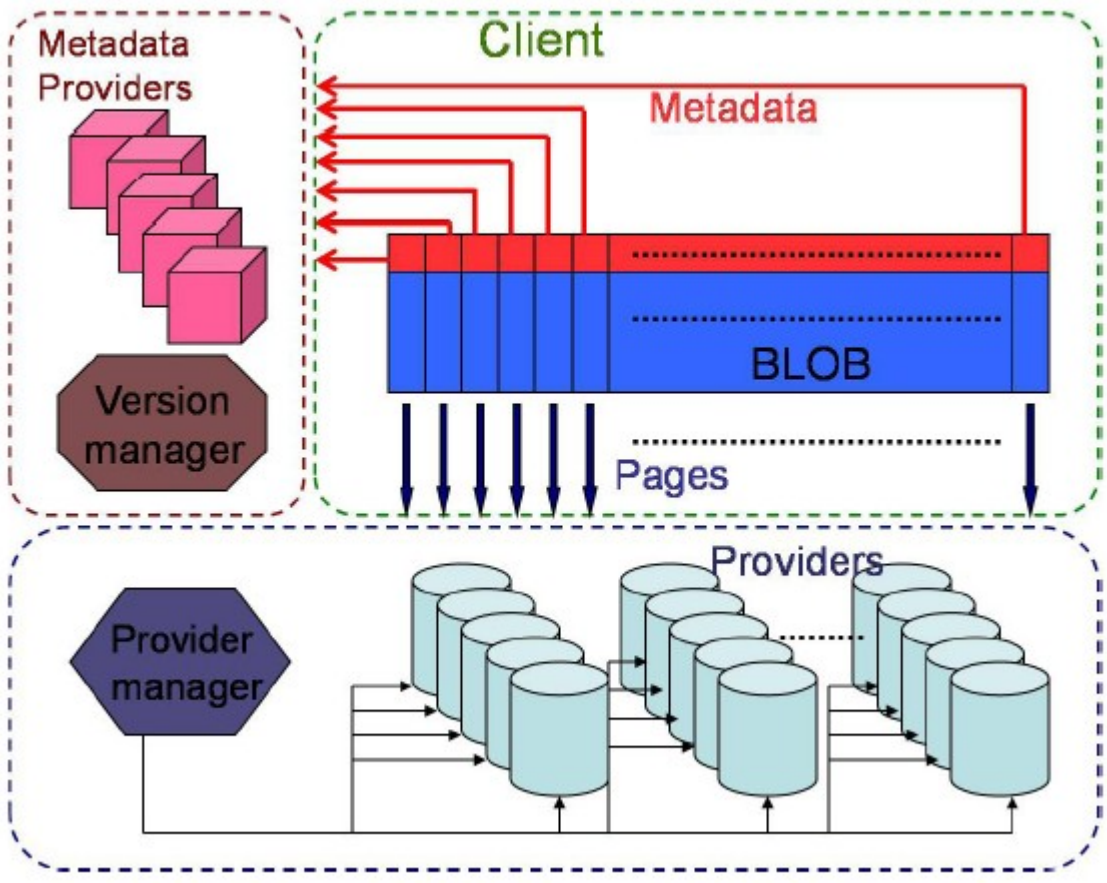
environment, it was built around a lock-free synchronization made possible by a version-oriented design. The entire data and metadata structures are decentralized enabling the BlobSeer system to provide a high data, parallel throughput.

The data itself has no structure. It is viewed only as flat sequence of bytes which can be reconstructed later at the user's request. Because it accepts fine-grain access to a large structure of data, this storage system is ideal for applications which use access patterns like Map/Reduce where workers usually access random and small portions of the data at hand.

Key design choices for BlobSeer are: versioning access interface, data striping, distributed metadata and lock-free, version-based concurrency control.

4.1 System Architecture

The BlobSeer components (actors) are represented by distributed processes which communicate between them using RPCs. There is no restriction on how many different processes can run on a



BlobSeer Architecture

single machine at the same time, so the entire system might as well be on the same physical machine (although one might not want this except for testing purposes).

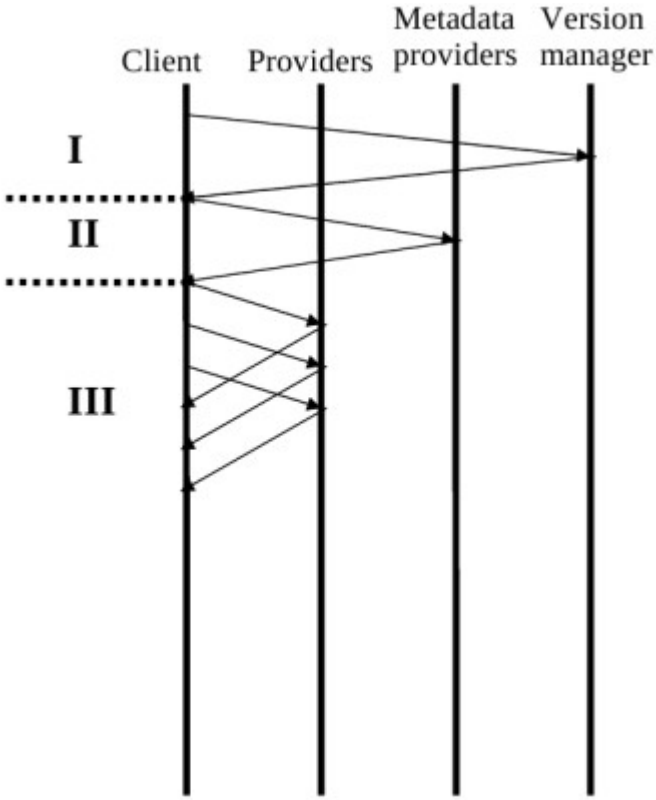
The Actors taking part in the system as seen in the image above are as follows:

- *Client* - a Client may issue one of the following commands:
  - CREATE - to create a new BLOB
  - WRITE - to write a new BLOB
  - APPEND - to write new data at the end of a BLOB
  - READ - to read back a BLOB
- *Data Provider (Provider)* - a Provider stores and manages the pages generated by WRITE/APPEND operations
- *Provider Manager* - a Provider Manager keeps informations about available Data Providers
- *Metadata Provider* - a Metadata Providers stores and manages informations about the data stored on Providers
- *Version Manager* - the Version Manager is the key actor of the entire system, registering data updates

The number of clients using the system may vary in time without the system knowing how many they are. BlobSeer supports multiple concurrent access so multiple clients can connect at the same time. Providers are free to join or leave the system whenever they want, keeping the entire system functional. The Provider Manager gets notified about any of the Providers' actions and will adapt the system as needed. Also, the Provider Manager schedules the placement of newly generated pages making use of a load balancing strategy of choice. Metadata Providers are responsible with keeping information about where to find specific data. The metadata information is kept in a distributed way so that the system does not suffer from a bottleneck when searching for metadata information. Last but not least, the most important actor of them all is the Version Manager where all the serializing of actions gets done. All the WRITEs and APPENDs operations pass through it, assigning a version number for each of the operations. Once the operation is completed the version becomes publicly available, thus guaranteeing total ordering and atomicity.

4.1.1 I/O operations

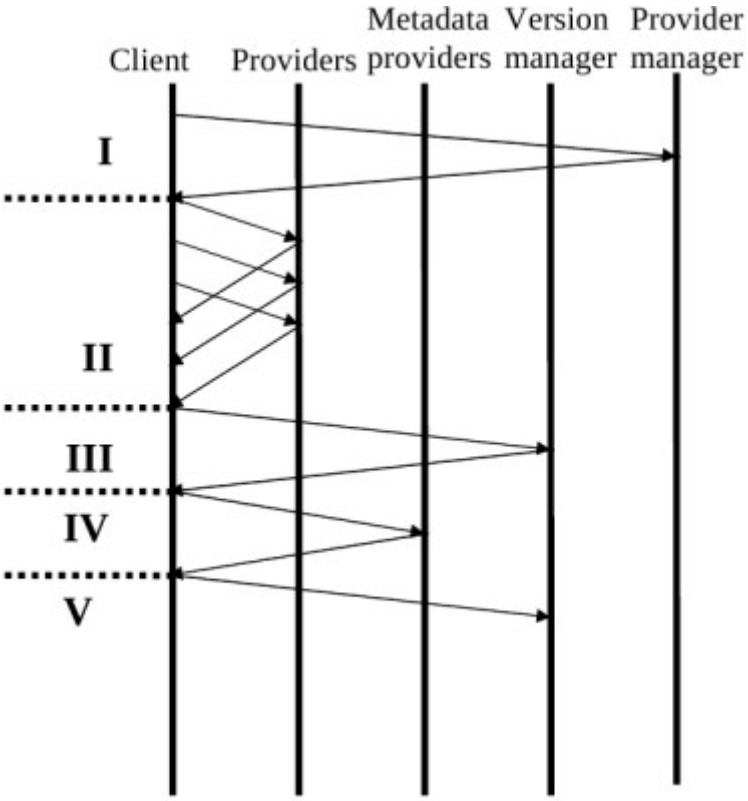
4.1.1.1 READ operation



How does a READ work for BlobSeer

Once the user requests a READ operation on a certain BLOB (the BLOBs are identified by a uniquer BLOB id), the client contacts the Version Manager. Providing the Version Manager with a BLOB id, a version number a range of pages needed, the manager will return and validate the request. Once the request is validated, the client directly queries the Metadata Providers to retrieve informations on where the pages requested are physically stored. Once they have the metadata, the client will fetch the requested BLOB pages in a parallel way.

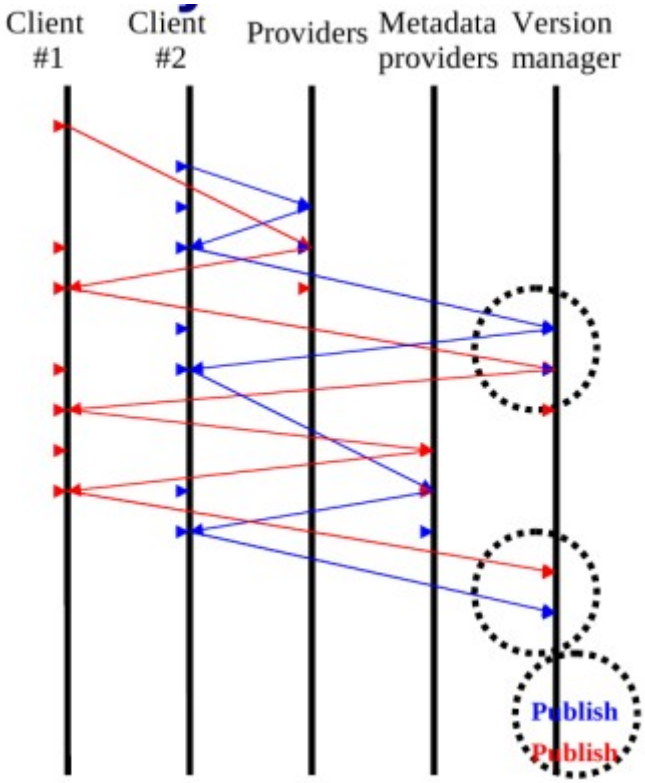
4.1.1.2 WRITE/APPEND operation



How does a WRITE/APPEND work for Blobseer

Once the user request a WRITE request, the client will contact the Provider Manager to obtain a list of providers that are available to take the requested load. The manner in which the Provider Manager will compute this list is at its sole discretion (using a load-balancing algorithm). However, the number of providers returned is usually fixed, one for each page of the BLOB the client needs to get written. Once this list is obtained, the client will write the corresponding pages to each provider in parallel. When the client receives an acknowledgement from all the providers, it contacts the Version Manager to obtain a version number based on which the metadata information is generated and stored on Metadata Providers. The client notifies the Version Manager of success and returns to the user. The final step is the Version Manager making the data available to the public by its given version number. The APPEND request is nothing more than a particular WRITE request and hence the way it works is basically the same.

4.1.1.3 Concurrency

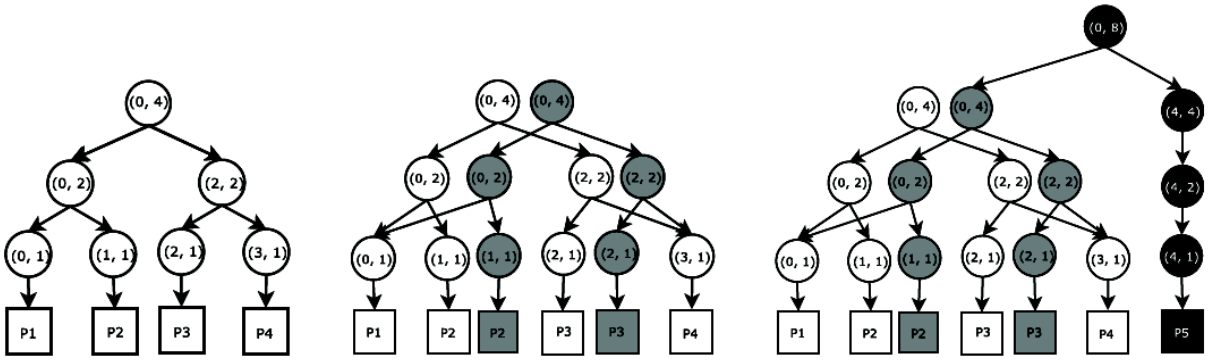


How does concurrency work for BlobSeer

As seen, every actor in the system is distributed, hence a high data throughput is guaranteed and with no bottleneck during the READ, WRITE or APPEND operation. One might argue that because the Version Manager is a single entity this might slow down the process of reading and/or writing BLOBs. However, by design, the Version Manager's actions are kept to be very simple and fast operations. Also, Version Manager is the one that actually serializes the concurrent access of the clients and because this is done *lock-free* it is actually much faster than any other way.

4.2 BlobSeer Key Design and Algorithms

4.2.1 Distributed Metadata



Metadata information

Accessing a BLOB from BlobSeer requires knowledge of the version number of the BLOB and the pages, given as an (offset, size) tuple, to be accessed. Metadata is required to transform the version number and the page range into the location of the actual physical data. The data structure used to hold such information is called *distributed segment tree*, one data structure for every version number in the system. This segment tree is actually no more and no less than a binary tree, the nodes of which contain the following information:

- if the node is not a leaf - contains a range of pages information that we will find under it
- if the node is a leaf - contains a pointer to the information regarding the page

Note that a leaf will always contain exactly one page and the root will cover the entire range of pages for an existing data object.

To perform very well under heavy concurrency, the tree nodes are distributed among the Metadata Providers in a DHT[2] (Distributed Hash Table). To avoid multiple computations and storage of the same information new metadata might use metadata from previous versions of the same BLOB (see the Metadata Information image above).

4.2.2 Data Striping

Obviously, large data objects that reach the size of Terrabytes or more cannot be stored in one chunk. This is why BlobSeer uses a process called *data striping* which consists of splitting the large object in smaller chunks that can be manageable. This chunks are called *pages* and have fixed size.

These pages are stored on different Data Providers, making use of a load-balancing policy implemented in the Provider Manager. This enables parallel uploading and downloading sustaining a high throughput of data in and out of the system.

4.2.3 Versioning Access Interface

The versioning part of the BlobSeer system is handled entirely on the client side. Each time a WRITE/APPEND operation is requested, a **new** snapshot is created rather than updating the old one. The metadata information is updated accordingly. The new snapshot will be accessed by an incremental version number so that older versions can still be accessed for an undefined period of time. The version number assigned are the system's responsibility and the clients must use the ones generated by the system.

Although a WRITE/APPEND operation generates a totally new version in the system, what is stored is only the difference between the old data and the new data, as well as between the old metadata and the new metadata. This saves both time and space on the long term. This design choice may lead to advance future features like branching, rollback, etc.

4.2.4 Lock-free Synchronization

Locking is rather a performance-killer and an error-prone method, this is why in BlobSeer synchronization is achieved through version-based concurrency control. This algorithm allows the number of parallel operations to be maximized. Avoiding synchronization at metadata or data level is achieved through a simple and straightforward technique: nothing gets deleted, only updated. One may notice that there is no DELETE operation in the BlobSeer system, only WRITE or APPEND.

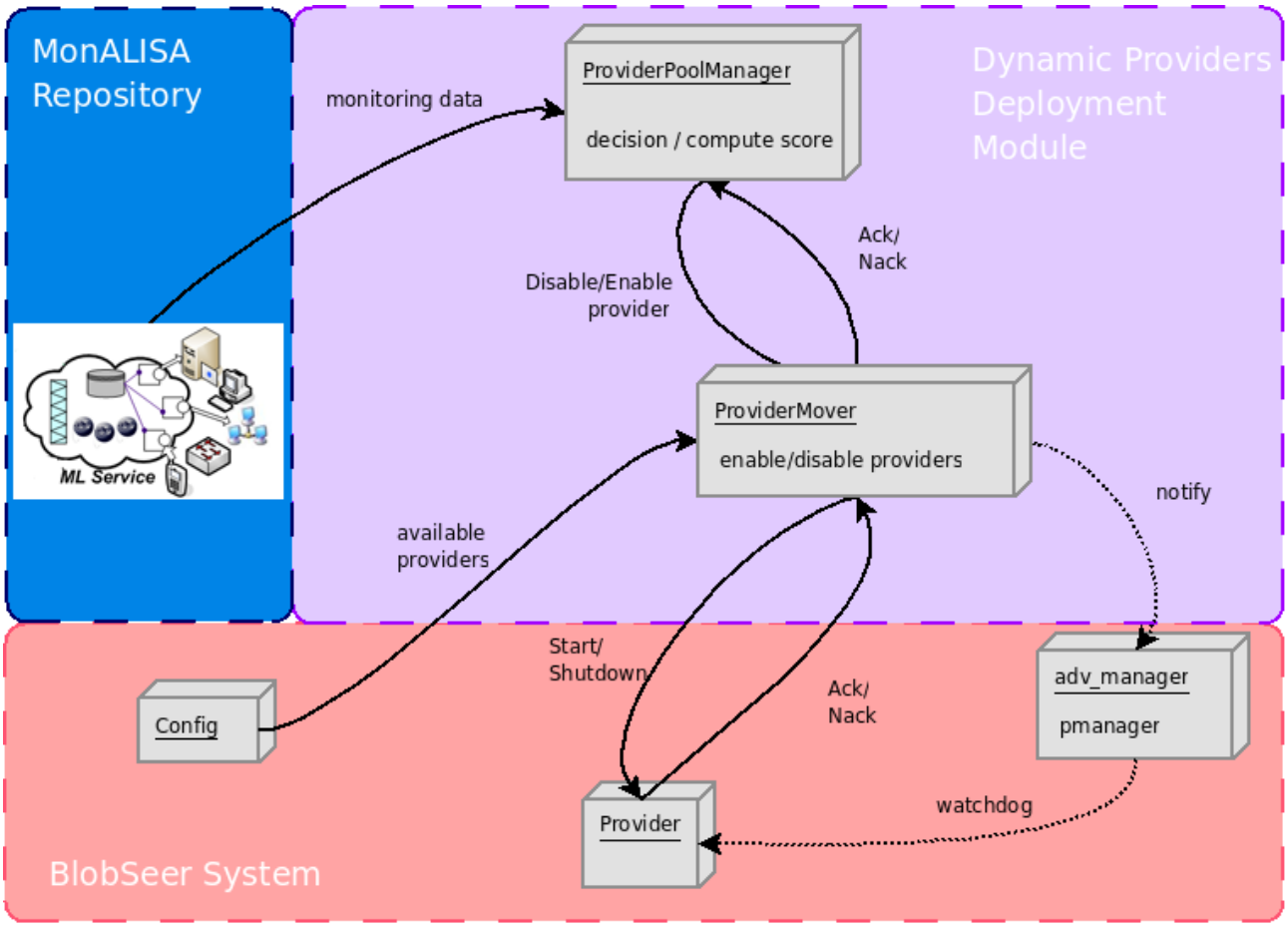
There are two phases: one, where the actual data block are written and the other where the version number is generated together with storing the new metadata information. In the first phase, there is absolutely no need for synchronization. The data can be written in a fully-parallel fashion enabling a high-throughput. In the second phase, the Version Manager generates the new version number which, in turn, helps generating the new metadata. The generation of the version number must be unique in the entire system, so this is the only moment in the usage scenarios where things are serialized by the Version Manager. Other than this step, the clients can do everything in a parallel manner.

5 Dynamic Deployment Module

5.1 Architecture Overview

Although the problem solved by the Dynamic Deployment module is rather complex, we tried to keep the architecture 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. Based on the score's value the main goal of this module can be achieved: take the decision if a provider is going to be removed or added from the active pool of providers.



System Diagram for Dynamic Providers Deployment Module

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 Provider Manager which can be turned on or off, depending on the decision taken. This part is also responsible with 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 Data Providers to see which ones are still alive, using a watchdog facility.

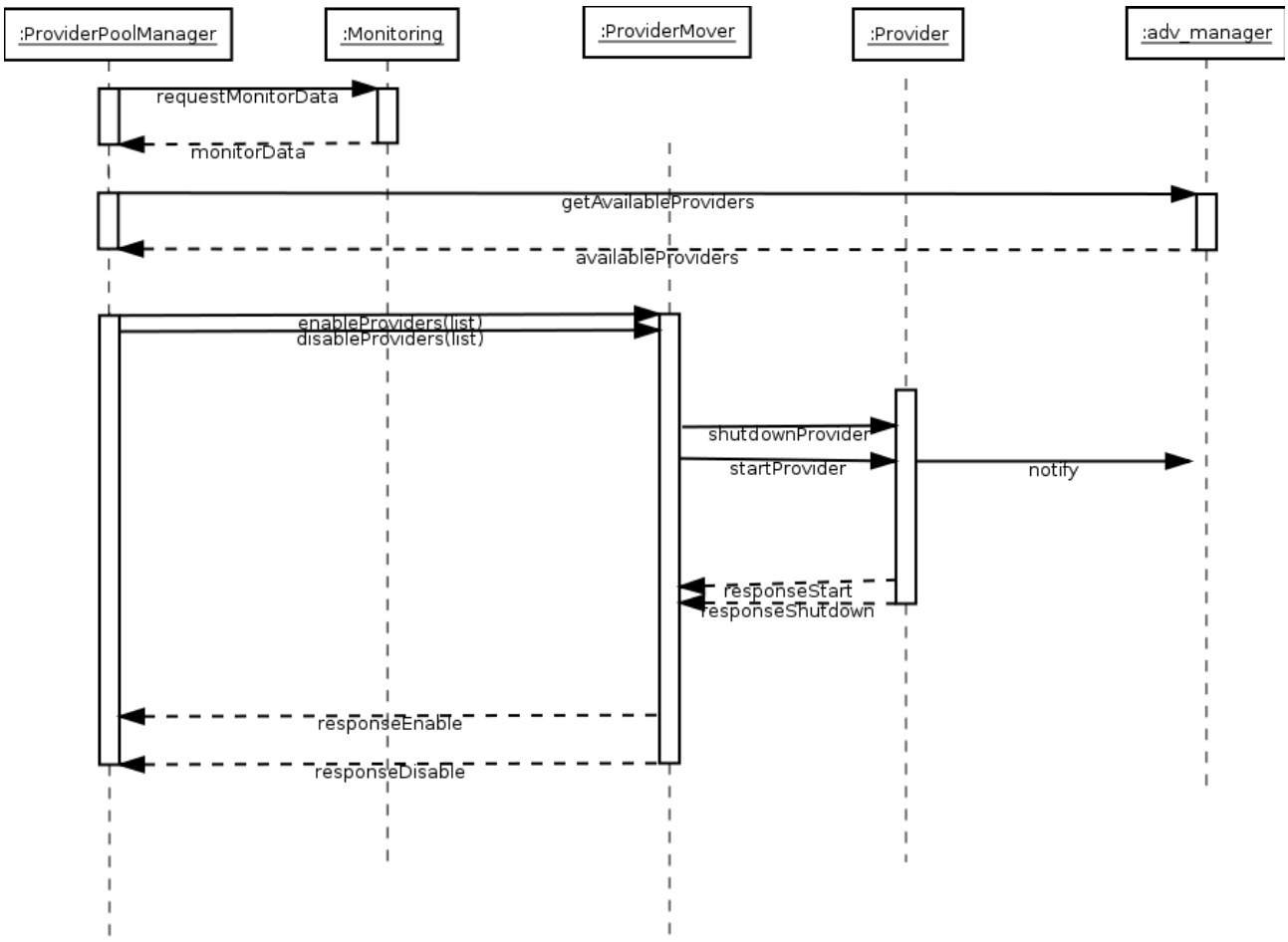
The main actors of the Dynamic Deployment module are the decision module and the action-taking module. *ProviderPoolManager* is the class responsible with the decision taking. This consists of enabling or disabling a number of Providers and it is based on the information given to the class by the monitoring module. The *ProviderMover* class is responsible with putting the decision into practice by moving a provider from the Active Pool of Providers to the Backup Pool of Providers or vice-versa, depending on what commands it receives from the *ProviderPoolManager*. These two main actors of our application interact closely with different other components, either custom (e.g.: monitoring module), or BlobSeer's (e.g. Provider Manager).

The interaction with BlobSeer's Provider Manager resumes to a RPC call. This allows our module to get a list of the actual active data providers running in the system at a specific moment in time. ProviderPoolManager reads the coordinates of the Provider Manager and contacts it through a RPC call. As a result, it will give back a list of tuples in the form of (host, port) where data providers are active. Once our application has this information it can proceed to analyzing every Provider based on the monitoring data provided.

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 which 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.



Sequence Diagram for Dynamic Providers Deployment Module

In the Sequence Diagram we can observe, step by step, the flow of the Dynamic Deployment module. First, the monitor data is retrieved in our custom database. This is done continuously, as a separate process by the monitoring module. Second, the ProvidePoolManager issues a RPC call to the Provider Manager (adv_manager class) to get the list of active Providers. Once we have the data, the Pool Manager can start computing the score for each one of the providers. Based on the configuration file and the heuristic, a decision is taken and communicated to the ProviderMover object. This, in turn, will call scripts that will start or stop a particular provider.

5.2 Heuristic Algorithm

5.2.1 Scenario Illustration

For an illustration on how the module will behave on certain patterns we will present a couple of examples of possible scenarios for the module's utility. They were chosen as they are very common on a daily base use and they leave no room for interpretations.

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 greater than 1
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. In the current implementation, shutting down the provider is simply a matter of calling an existing script; however, it is important for the module to check the data's 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 makes the necessary checks.

5.2.2 Computing the Score

The scoring algorithm, in its current form, provides a method to detect which **Providers** should be moved from **APP** to **BPP**.

5.2.2.1 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 behavior

Physical factors to be considered:

- Free Disk Space (f)
- Average bandwidth 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)

5.2.2.2 Example of scenarios

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

1. **f** >= 70%, **d** > 1 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

5.2.2.3 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	>= 60% (avg over 6h)	0
			>= 10% && < 60% (avg over 6h)	0.5
			< 10% (avg over 6h)	1

5.2.2.4 The final score

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

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

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*

5.2.2.5 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, leave the **Provider** in **APP**
2. else, move the **Provider** from **APP** to **BPP**

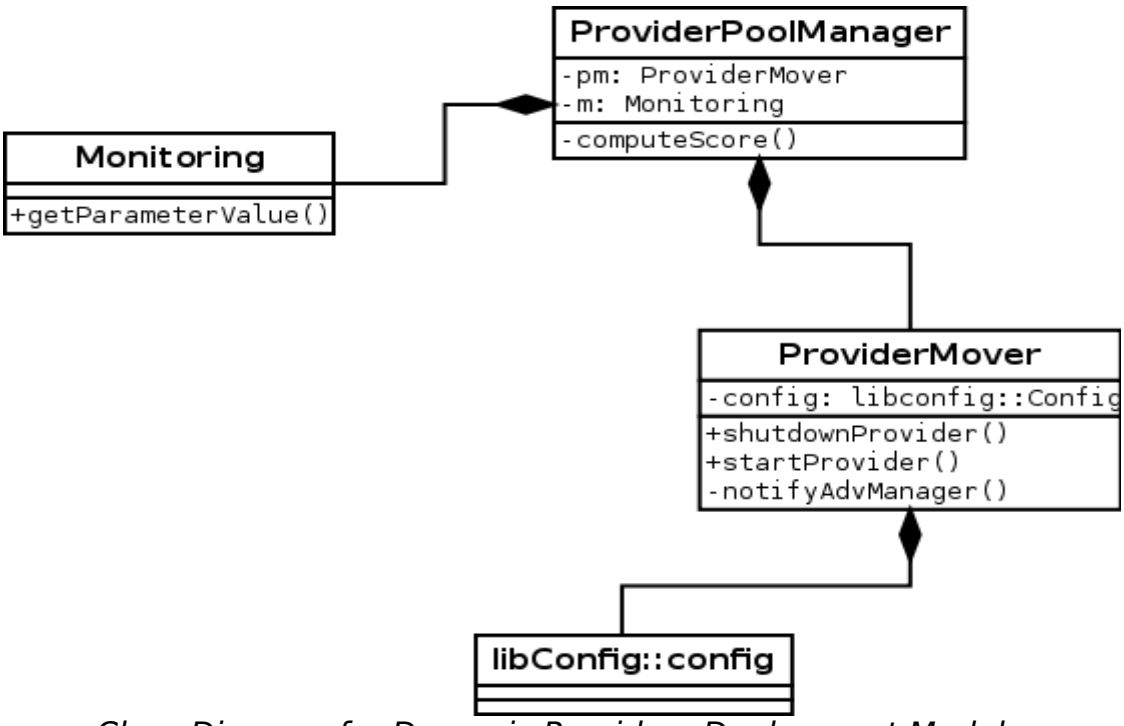
6 Implementation

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 even close to being neither 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 which Providers to get 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 . Currently, there is no common interface which can provide the application with all the data it needs in order to take the decisions. We are therefore building this interface for all the read operations. This interface is responsible with retrieving data from both sources described in the monitoring data section. In the background, it has to manage a couple of database connections, refresh them as needed and use them as optimal as possible.

We will detail every aspect of the implementation in the following sections, however to have an airplane view of the modules and classes that are expected to be present one can look at the following diagram:



Class Diagram for Dynamic Providers Deployment Module

6.1 Scenarios Specification

As described in the Scenario Illustration section there are a lot of factors and, thus, patterns that might emerge which have to be taken into consideration. There is a need for the specification framework to be flexible and easy-extensible. We have therefore decided to take the most general approach 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 put together in a scenario that finalizes with giving a score. This score maybe the most important value in 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.

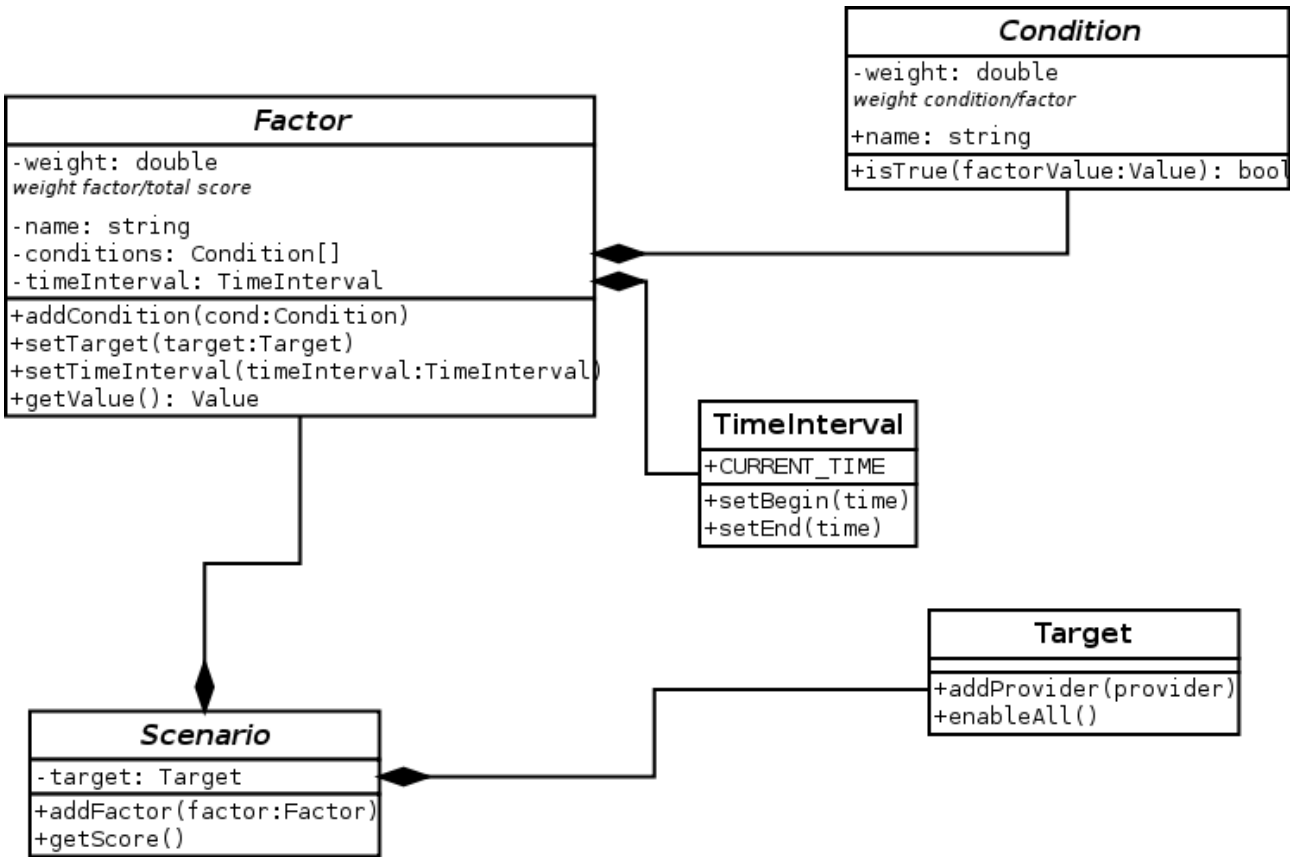
The following configuration section presents how one or multiple scenarios can be created just by editing a configuration file without a need to write a single line of code. The idea is simple: every scenario has one or more factors to check, every factor has a series of conditions which should be checked. Once a condition is true then the weight of it is to be considered. Of course, it has more flexibility and provides the possibility to change the implementation for a factor or a condition with a more specific one in case one needs it.

```
# ...
specifications: {
   # Targets that should compute our factor on
   #   e.g.: "every", "all", "192.168.1.1"
   targets: {
     t1: "every";  # each provider in the system
     t2: "all";    # all providers in the system
   };

   intervals: {
     i1: (600);             # 10 minutes earlier to now
     i2: (600, -1);         # the same --> -1 = now()
     i3: (600, 300);        # 10 minutes earlier to 5 minutes earlier
   };

   # Conditions that might be satisfied by a factor
   conditions: {
     c1: {
       upper = 65.0;
       lower = 63.0;
       weight = 0.1;
       implementation = "condition_general";
     };

     # Free space conditions
     free_c1: {
```

```
    upper = 1.0;
    lower = 0.7;
    weight = 0.6;
    implementation = "condition_general";
  };
  free_c2: {
    upper = 0.69;
    lower = 0.4;
    weight = 0.35;
    implementation = "condition_general";
  };
  free_c3: {
    upper = 0.39;
    lower = 0.0;
    weight = 0.05;
    implementation = "condition_general";
  };

  # Reads/Writes per time unit conditions
  rws_c1: {
    upper = 0.99;
    lower = 0.0;
    weight = 0.7;
    implementation = "condition_rws";
  };
  rws_c2: {
    upper = 2.99;
    lower = 1.0;
    weight = 0.2;
    implementation = "condition_rws";
  };
  rws_c3: {
    upper = 99999.9;
    lower = 3.0;
    weight = 0.1;
    implementation = "condition_rws";
  };
};

# Factors that can be combined in a scenario
factors: {
  # Free space
  free_space: {
    # The sum of conditions' weight MUST be 1
    conditions: ("free_c1", "free_c2");
    # The name should be in the db if using a general implementation
    name: "free";
    # "physical" / "internal"
    type: "physical";
    weight: 0.6;
    interval: "i1";
    implementation = "factor_general";
  };

  # Reads per unit of time
  reads_per_time: {
    conditions: ("rws_c1", "rws_c2", "rws_c3");
    name: "blob_id";
    type: "internal";
    weight: 0.2;
```

```
        interval: "i1";
        implementation = "factor_read";
      };

      # Writes per unit of time
      writes_per_time: {
        conditions: ("rws_c1", "rws_c2", "rws_c3");
        name: "blob_id";
        type: "internal";
        weight: 0.2;
        interval: "i1";
        implementation = "factor_write";
      };
    };

    scenarios: {
      s1: {
        # The sum of factors' weights MUST be 1
        factors: ("free_space",
                  "reads_per_time",
                  "writes_per_time");
        target: "t1";
      };
    };
  };
  # ...
```

The actual architecture for the specification framework is presented next. In order to obtain the flexibility we presented before, we have chosen to implement it as follows:



Scenario Implementation Architecture

Details about the classes:

- *Target*: The Target class specifies a Provider, a set of Providers or all available Providers to be the target of the Factor to be considered
- *TimeInterval*: The TimeInterval class specifies a time span between the Factor that is considered to be computed
- *Condition*: A Condition is an interface which should have different implementations of the *isTrue* method depending on the factor. Each factor will have multiple Condition instances in order to fit the Factor value with a certain Condition weight. All conditions added to a Factor should be mutually exclusive. If they are not, no error will be raised but one should bear in mind that the first condition (in the exact order they were added to the Factor instance) encountered as true is the one to be considered.
- *Factor*: A Factor is an abstract class and there are different specifications for each Factor the module will consider for the *getValue()* function. This implementation will be responsible with getting the value needed to evaluate the Condition instance.
- *Scenario*: The Scenario class will we responsible with allowing a common interface for specifying the Factors with each of their Targets, TimeInterval and Conditions. The *getScore()* method will allow the user to get the final score for this scenario.

6.2 Monitoring Module

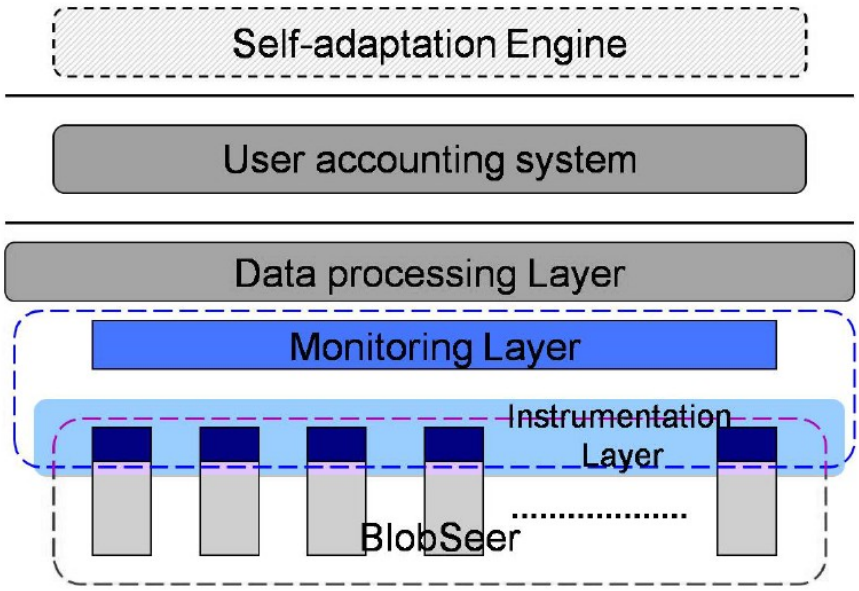
The Monitoring data module is one of the most important components of our application. One might notice that without correct and real time monitoring data, our heuristic algorithm is of no use.

First of all, there are two kinds of data we will need to collect: Internal data and Physical data. Internal data describes monitoring values for the BlobSeer system and evaluates Providers in terms of: number of reads, number of writes, degree replication, etc. Physical data describes monitoring values for the actual physical machines on which the provider processes run. The metrics being evaluated include, but are not limited to: CPU, free space, bandwidth usage, etc. Obtaining these values is a complex task by itself so we will present how we manage it.

The design for this module was created to have a single interface for both type of values. Thus we

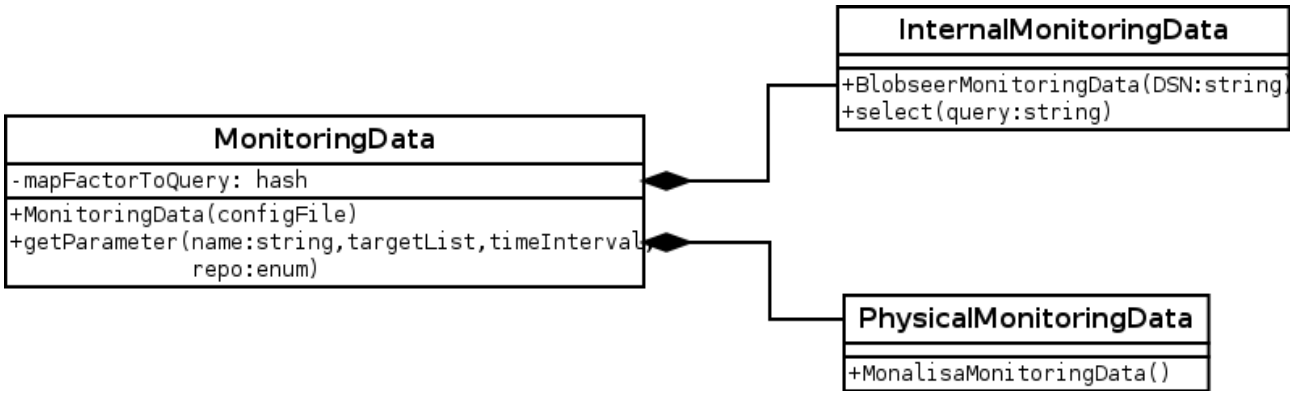
have chosen to create a common database using the PostgreSQL[10] database system. Every Provider will have three tables in this custom database: two for Internal factors (one for the reads and one for the writes) and one for the Physical factors. The tables are not complex, they just record a timestamp and all the mandatory factors. This database design was chosen keeping in mind that a huge amount of records are going to be inserted in a short period of time. Splitting the tables' structure as we mentioned before, we are able to make only simple queries over the database with no inner joins required, allowing fast response of the monitoring module.

Getting these values in our custom database requires a two-stage process, one stage for each type of monitoring data. For Physical factors this stage describes as more or less an easy step because the machines that are monitored by MonALISA are already sending the physical information that we need in our heuristics. This means we only need to capture this data using a filter and to make it persistent in our custom database. However, for the Internal factors, things are not that easy anymore. Luckily, an already made User Accounting[4] module will help us here. This module implements the ApMon interface and sends the specific, BlobSeer internal data, to the MonALISA repository. Again, all we have to do is to create a filter and save it in a database along with the Physical factors.



User Accounting module architecture

Monitoring data module will have one common interface with two different implementations, one for each kind of monitoring data. You can see the module's components in the following diagram.



Monitoring Module Architecture

Details about the classes:

- *MonitoringModule*: This class is the entry point for this module and abstracts all calls to the two repositories. It has one main method, getParameter() which allows the user to get the value of a parameter specified by its name.
- *InternalMonitoringModule*: This class handles all BlobSeer-specific parameter requests
- *PhysicalMonitoringModule*: This class handles all physical factors' values request

6.3 Configuration

The Configuration component is implemented using the libconfig[11] library which is a C/C++ Configuration Library. The choice for this configuration library was made because the BlobSeer implementation already uses it while it is a simple, robust and lightweight component. Using this library we were able to create thorough configuration file for the Dynamic Deployment module.

To make the integration easier with the BlobSeer system we used the same configuration file as the system uses for the Provider Manager, Version Manager, etc. All options related to the current module are under a sectioned named *dd*. As it can be seen in the full configuration file in the Appendix at the end of this paper, there are options to customize the start and kill scripts for providers, the repository, the thresholds for the computed score and other options. However, the

most important configuration option in this file is the possibility to specify different scenario without writing any bit of code provided that one does not need specific data. Everything can be specified in the configuration file: time intervals, targets, conditions, factors and how all of them are glued together to form a scenario on which a decision will be made. As long as a general implementation is enough for the conditions and factors one might need, no code must be written. For more information about time intervals, targets, conditions, factors and scenarios you can read Scenarios Specification section above.

7 Experimental Results

We have conducted several experiments in order to determine how intrusive our system is, how much bandwidth it consumes and what could be the maximum time to put in practice an action based on a real conditioned appeared in the system.

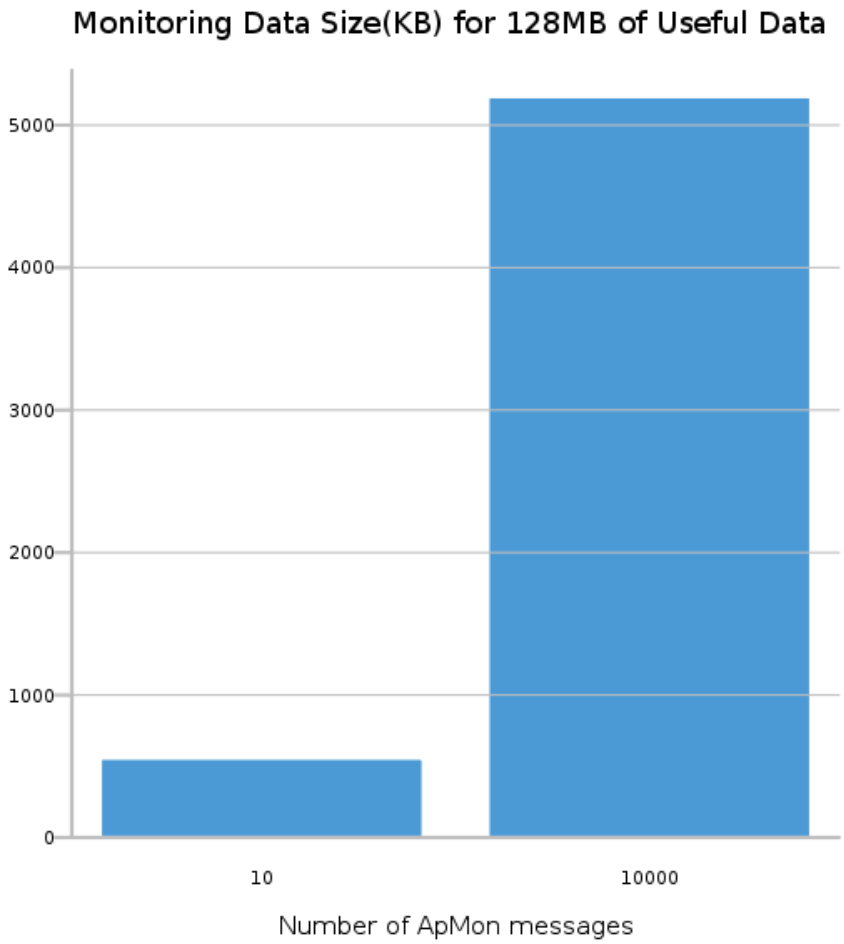
The testing infrastructure consisted of two physical machines and three virtual ones. Four of them were present on a LAN, while the other one was connected through the Internet. The machine outside the local network held the MonALISA repository and the communication with it was made over a roughly 0.5-1.0MB Internet connection. The physical machine on the LAN was used to host a Provider Manager, Metadata Manager and a Version Manager. The other three were used to deploy 1-100 providers depending on the test conducted.

7.1 Bandwidth Usage Test

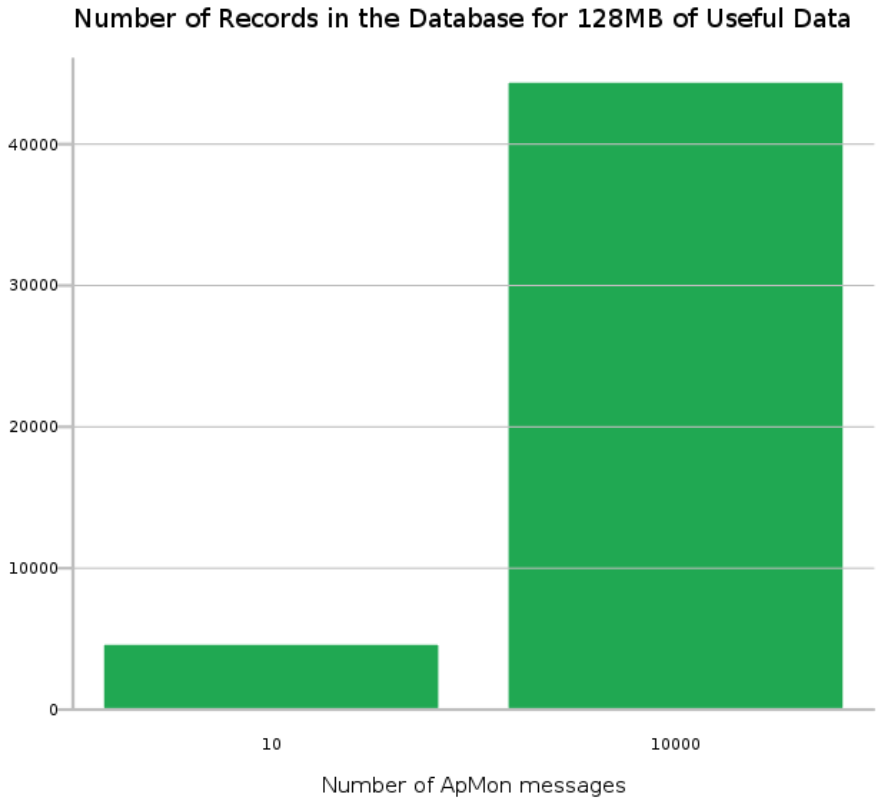
We enabled monitoring for the BlobSeer system and experimented with WRITE and READ operations over BLOBs in the system. Every operation is recorded by the MonALISA's ApMon interface and is sent to a repository we set up. Using the MonALISA's detailed logging capabilities we were able to measure the packages' length. The ApMon interface permits changing how much of the monitoring data should be sent to the MonALISA repository. This represents the sampling rate or the number of messages per second and can be changed from a configuration file.

We started conducting the test with a maximum of 10000 messages per second. As we expected this would enable a high throughput of monitoring data and huge bandwidth usage because every WRITE or READ operation would be sent to the central repository. The data obtained meets our

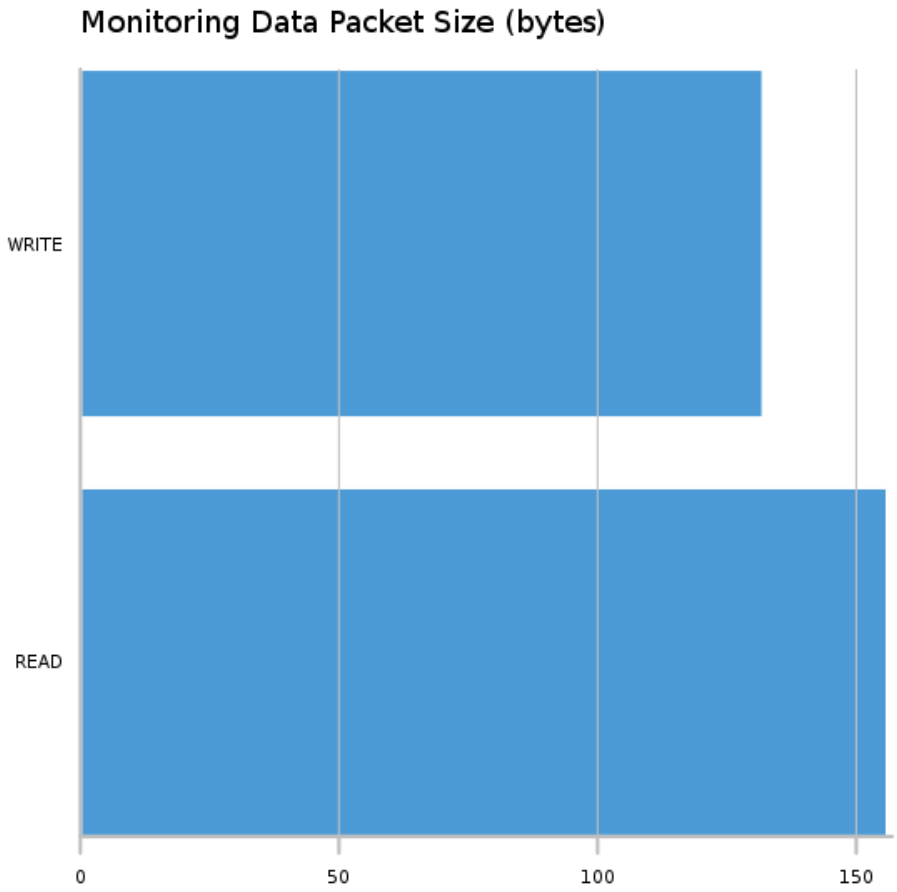
expectations - 5MB of monitoring data for 128MB of useful data or 4% overhead. Given the fact that BlobSeer is used for large data needs, we could easily end up with 40GB of monitoring data for just 1TB of useful data. This cannot be acceptable.



We switched to test with a maximum of 10 messages per second. Given our module's needs, 10 messages per second is more than enough to take a representative action in a timely manner. Our monitoring data storage needs were cut down by a 1000 factor, yielding 5KB of data for 128MB, or 0.004% overhead.



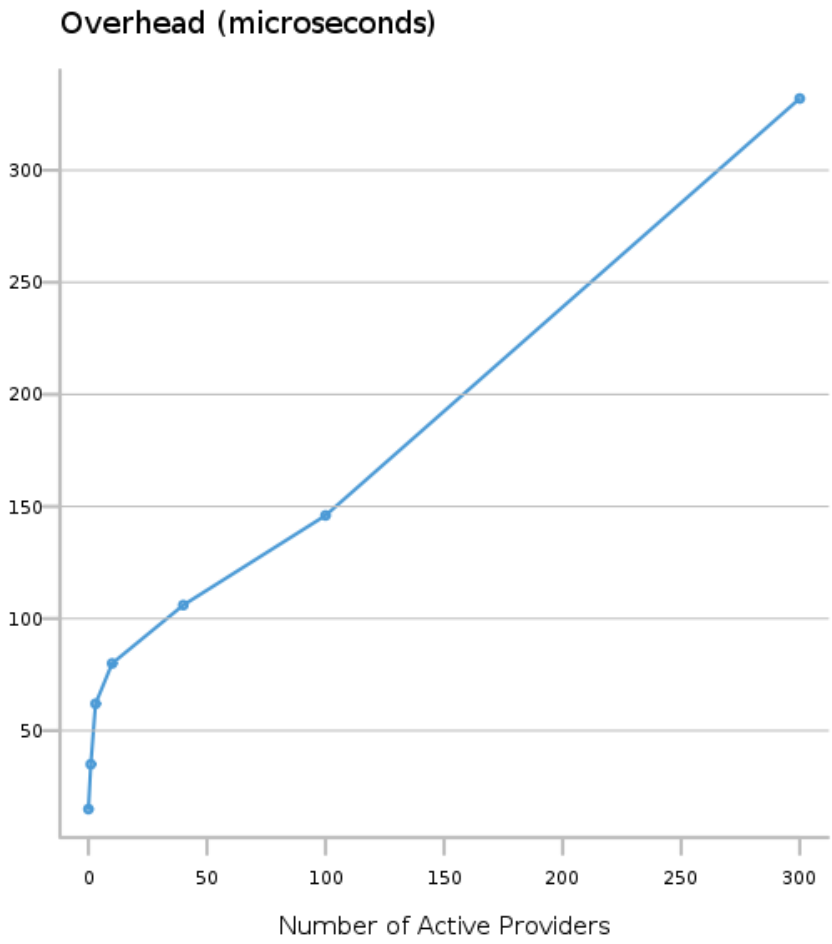
Analyzing the bandwidth usage, we can add to those 5KB of monitoring data the size of the UDP header for each of the packet sent (ApMon uses UDP packages to send the monitoring data). The 5KB are sent in approximately 4400 UDP packages, each package having an 8 bytes overhead for the UDP header. This adds up to 35KB of bandwidth usage for 128MB of real data. These tests also revealed the fact that READ packages are 18% larger than the WRITE packages (due to the different information sent for each of the packages).



READ vs. WRITE Monitoring Packet Size

7.2 Intrusiveness Test

The Dynamic Deployment module was constructed so that it should not be too intrusive to the BlobSeer system. As described in the architecture, the only communication done with BlobSeer is when getting a list of active providers. Thus, the only time spent in getting this list of active providers is actually the time that represents a quantification for our module's intrusiveness, next to the BlobSeer system. The following graph represents this time.



Overhead for Provider Manager

This entire time is represented as an overhead only for the Provider Manager. Our module makes an RPC call to the Provider Manager and gets the list of active providers. In the above graph, the time spent on the network is not included, as we wanted to measure the time we keep the Provider Manager busy and not the time it gets the module to actually receive the list. Another note to the above result is that, under heavy load for the Provider Manager (lots of READ/WRITE operations from different providers), the total time for receiving the complete list might increase. This is because the internal Provider Manager list is protected by a lock, thus lock contention time may vary. However, this is not an overhead for the Provider Manager and, again, it is not included in the above graph.

7.3 Latency Time to Action

We will define for the purpose of this section the *latency* time as the difference between the time when an event actually happens on the real physical machine and the time it will take to trigger the corresponding decision and action inside our module.

It is obvious that a lot of factors shall interfere when trying to compute the latency. The Dynamic Deployment module communicates several other systems, thus the time can vary greatly. A formula for the total time can be given and we can analyze every factor of it. However, extensive testing should be done in real distributed systems in order to obtain valid statistical data.

The total latency time is a sum of multiple times:

- the time it takes for the ApMon interface to detect the condition and send the monitoring data to the MonALISA repository (t_{mon})
- the configuration time for the **DD** module that specifies the waiting between checking cycles (t_{wait})
- the time to receive a complete list of active providers from the Provider Manager (t_{list})
- the time to analyze every provider ($t_{analyze}$)
- the time to connect to the provider and take the action ($t_{provider}$)

Thus, the final formula we will give as the total latency time is:

$$t_{latency} = t_{mon} + t_{wait} + t_{list} + t_{analyze} + t_{provider}$$

However, one should note that nothing in this formula includes network time which could considerable increase the total latency time.

The t_{mon} factor depends on how fast the ApMon interface checks and sends the new monitoring data. Modifying this parameter in order to obtain a time that tends to be 0 is feasible, still, most likely, not wanted. As we previously discussed in the Experimental Results section, under the Bandwidth Usage Test one can quickly obtain a high overhead in the monitoring data if we choose to increase the rate in which we sample the system.

The t_{list} factor depends on the number of active providers in the list. A discussion has been made

about this time in the Experimental Results section, under Intrusiveness Test. However this time is more likely to become non-important against the network time, which again this is not included in our formula.

The $t_{analyze}$ factor depends on the number of factors and conditions a scenario is made of. Scenarios, factors and conditions are described in detail under the Implementation section and can be easily added, removed or changed in the configuration file. This is why this time is hard to estimate when it comes to different deployments on different systems. Moreover, the time is dependent on the size of the monitoring data database and how soon a certain condition for a certain factor is met. The number of factors also modifies this time drastically as, almost always, a factor specification will hit the database at least once. Taking an example of a scenario with two factors, each of them with three conditions to be checked, a time of ~0.4 seconds is needed to take a decision on a small database.

The $t_{provider}$ factor depends on the scripting involved when modifying a provider status. In the current implementation, the shutting down of a provider on a physical machine boils down to creating a secure connection through SSH to the machine and sending a signal to the provider. This is a trivial implementation and which is dependent only on the network time. However, in the future, this time might increase as one could write a more complex script for modifying the providers' status.

The t_{wait} factor is probably the one that makes all the difference for our latency measurement. One can choose to configure the Dynamic Deployment module to not at all between checking cycles. This would make the module to continuously check the new monitoring data received and take an action based on that. However, second-precision might not make sense for the module's main use case so most likely the configuration would made so that an interval of at least some seconds before re-starting a check cycle. If this parameter is made large enough (e.g.: might be of order of minutes, depending on the system deployment needs) the discussion about other times becomes obsolete.

8 Conclusion

In this paper we described a module which brings the power of self-adaptedness to any distributed storage system, analyzing the specific case of implementation for BlobSeer. We are confident that

this module will bring great power in the context of Cloud Computing bringing flexibility and, most important, cost-reduction capabilities for the system. To be able implement the module and its decision-taking algorithm we used monitoring data from a MonALISA repository built with custom filters. We tested our module in order to see how intrusive it is and what is the latency time to action revealing conclusive results.

8.1 Future Work

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 pluggable 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 one might think about improving the system's behavior when the pool of active nodes is modified.

The Heuristic algorithm should be careful monitored in production. Experimental results during testing revealed that it is good enough an algorithm, however we do not think it is yet valid for a production environment. Moreover, monitoring data should be expanded to include various factors, both internal and physical in order for the heuristic to become better and better. For example, a future direction in this area can be analyzing the actual costs that go together with the shutdown of a provider, costs like the ones related to moving the data and replicating it somewhere else.

8.2 Credits

Special thanks goes to Alexandru Costan, the person who steered me on the right path and who always has a good advice to give. Thank you!

Thanks also goes to Alexandra Carpen-Amarie for all the help with the monitoring module and to Alexandra-Catalina Anton for the English review.

9 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. Distributed Hash Table: http://en.wikipedia.org/wiki/Distributed_hash_table
3. MonALISA: http://monalisa.cacr.caltech.edu
4. Mihaela-Camelia Vlad in the paper Distributed Monitoring for User Accounting in BlobSeer Distributed Storage System
5. Google File System - http://labs.google.com/papers/gfs-sosp2003.pdf
6. Hadoop File System - http://hadoop.apache.org
7. Amazon's Simple Storage System - http://aws.amazon.com/s3/
8. DeepStore - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.5241&rep=rep1&type=pdf
9. Bogdan Nicolae, Luc Bouge, Gabriel Antoniu in the paper BlobSeer: dealing with increasing storage demands of large-scale data-intensive distributed applications
10. PostgreSQL relational database. http://www.postgresql.org/
11. libconfig, C/C++ Configuration File Library: http://www.hyperrealm.com/libconfig/

10 Appendix

10.1 Code Snippets

Full source code at http://svn2.xp-dev.com/svn/blobseer/

10.1.1 Configuration file

```
### Configuration options for the Dynamic Deployment module ###
dd: {
    max_cycles = 1;                  # how many successful iterations
    sleep_between_cycles = 0;        # seconds
    sleep_no_active_providers = 0;   # seconds

    scripts: {
      start: "scripts/start.sh";
      stop: "scripts/stop.sh";
    };

    repo: {
          db: {
            driver = "postgresql";
            name = "test";
            host = "localhost";
            port = "5432";
            user = "root";
            pass = "12345";
          };
      };

     # Limits above (add) and below (remove) to which we'll take actions
     # agains the number of data providers in the pool
     thresholds: {
        remove = 0.6;
        add = 0.2;
     };

     specifications: {
        # Targets that should compute our factor on
        #   e.g.: "every", "all", "192.168.1.1"
        targets: {
          t1: "every";  # each provider in the system
          t2: "all";    # all providers in the system
        };

        intervals: {
          i1: (600);            # 10 minutes earlier to now
          i2: (600, -1);         # the same --> -1 = now()
          i3: (600, 300);        # 10 minutes earlier to 5 minutes earlier
        };

        # Conditions that might be satisfied by a factor
```

```
        conditions: {
          c1: {
            upper = 65.0;
            lower = 63.0;
            weight = 0.1;
            implementation = "condition_general";
          };

          # Free space conditions
          free_c1: {
            upper = 1.0;
            lower = 0.7;
            weight = 0.6;
            implementation = "condition_general";
          };
          free_c2: {
            upper = 0.69;
            lower = 0.4;
            weight = 0.35;
            implementation = "condition_general";
          };
          free_c3: {
            upper = 0.39;
            lower = 0.0;
            weight = 0.05;
            implementation = "condition_general";
          };

          # Reads/Writes per time unit conditions
          rws_c1: {
            upper = 0.99;
            lower = 0.0;
            weight = 0.7;
            implementation = "condition_rws";
          };
          rws_c2: {
            upper = 2.99;
            lower = 1.0;
            weight = 0.2;
            implementation = "condition_rws";
          };
          rws_c3: {
            upper = 99999.9;
            lower = 3.0;
            weight = 0.1;
            implementation = "condition_rws";
          };
        };

        # Factors that can be combined in a scenario
        factors: {
          # Free space
          free_space: {
            conditions: ("free_c1", "free_c2");      # The sum of conditions'
weight MUST be 1
            name: "free";                            # This should be in the
database if using a general implementation
            type: "physical";                        # "physical" / "internal"
            weight: 0.6;
            interval: "i1";
```

```
            implementation = "factor_general";
          };

          # Reads per unit of time
          reads_per_time: {
            conditions: ("rws_c1", "rws_c2", "rws_c3");        # The sum of
conditions' weight MUST be 1
            name: "blob_id";                                   # Can't be
anything here as long as it's valid
            type: "internal";                                  # "physical" /
"internal"
            weight: 0.2;
            interval: "i1";
            implementation = "factor_read";
          };

          # Writes per unit of time
          writes_per_time: {
            conditions: ("rws_c1", "rws_c2", "rws_c3");        # The sum of
conditions' weight MUST be 1
            name: "blob_id";                                   # Can't be
anything here as long as it's valid
            type: "internal";                                  # "physical" /
"internal"
            weight: 0.2;
            interval: "i1";
            implementation = "factor_write";
          };
        };

        scenarios: {
          s1: {
            factors: ("free_space",                  # The sum of factors'
weights MUST be 1
                      "reads_per_time",
                      "writes_per_time");
            target: "t1";
          };
        };
     };
};
```

10.1.2 Monitoring interface

```
typedef pair<boost::posix_time::ptime, double> record_t;
typedef list<record_t *> list_record_t;

/**
 * Interface for specifying the get_parameter method for the repositories
 */
class monitoring_data {
public:
    monitoring_data(libconfig::Config *config);

    monitoring_data_result* get_parameter(const string &param);
    void get_critical_providers(
            boost::unordered_set<string> *critical_providers);
    void set_targets(list<string> &targets);
    void set_interval(boost::posix_time::ptime &start,
```

```
            boost::posix_time::ptime &end);
    void set_aggregate(int agg_type);

    virtual ~monitoring_data();
protected:
    /**
     * Initialize database & other (lazy)
     * @see init_
     */
    bool initialize_();

    /**
     * Builds a template for SELECTs
     */
    string build_query_template_();

    /**
     * In the future, might need to specialize this function for each
     * of it's subclasses
     */
    virtual bool get_parameter_specific_(const string &param,
            monitoring_data_result *result);

    /**
     * Specific for each of the implementations
     *
     * @param returns the list of table names from which we should get data from
     */
    virtual void build_table_names_(list<string> *tables);


    libconfig::Config *config_;
    // Lazy initialization
    bool init_;
    // Connection to the database
    db_connection *conn_;

    list<string> targets_;
    time_t start_, end_;
    int agg_type_;
    int op_type_;
};

/**
 * Wrapper class for the results brought from the two repositories. Some
 * fields make sens only for one type of repository
 *
 * About ownership of this class
 *  - this class is usually alloc'd by get_parameter()
 *  - the caller of this method has the ownership over the object created
 *  - do *NOT* free the record list/pair manually, just make sure you free
 * the instance once your done, if it was alloc'd manually
 */
class monitoring_data_result {
public:
    /**
     * Add a record for this result instance
     */
    void add_record(boost::posix_time::ptime rectime, double value);
```

```
    /**
     * Simple iterator accessors
     */
    list_record_t::const_iterator get_begin() {
        return this->records_->begin();
    }

    list_record_t::const_iterator get_end() {
        return this->records_->end();
    }

    unsigned int get_count() {
        return this->records_->size();
    }

    monitoring_data_result();
    virtual ~monitoring_data_result();
private:
    // List of entries  (record time, record value)
    list_record_t *records_;

};

/**
 * Specific implementation for the Internal repository
 */
class internal_monitoring_data : public monitoring_data {
public:
    internal_monitoring_data(libconfig::Config *config);
    void set_operation_type(int type);
    virtual ~internal_monitoring_data();
private:
    void build_table_names_(list<string> *tables);

};

/**
 * Specific implementation for the Physical repository
 */
class physical_monitoring_data : public monitoring_data {
public:
    physical_monitoring_data(libconfig::Config *config);
    virtual ~physical_monitoring_data();
private:
    void build_table_names_(list<string> *tables);

};
```

10.1.3 Unit testing for Scenario implementation

```
BOOST_AUTO_TEST_SUITE(factor_test)
BOOST_AUTO_TEST_CASE(general_usage_test) {
    factor_general f;
    condition_general *c1, *c2;
    boost::scoped_ptr<monitoring_data_result> result;
    libconfig::Config config;
    target t;

    const double lower = 0;
```

```
    const double upper = 200000;
    const weight_t w = 0.9;
    const weight_t wrong_w = 1.3;
    const string config_path = "../test.cfg";

    BOOST_CHECK_THROW(f.set_weight(wrong_w), dd_exception);
    BOOST_CHECK_NO_THROW(f.set_weight(w));
    BOOST_CHECK_CLOSE(f.get_weight(), w, 10e-10);

    c1 = new condition_general();
    c1->set_lower_limit(lower);
    c1->set_upper_limit(upper);
    c2 = new condition_general();
    c2->set_lower_limit(lower);
    c2->set_upper_limit(upper);

    // c1 & c2 becomes property of f
    config.readFile(config_path.c_str());
    f.set_config(&config);
    f.add_condition(c1);
    f.add_condition(c2);

    result.reset(f.get_value(t));
    for (list_record_t::const_iterator it = result->get_begin();
            it != result->get_end(); ++it) {
        record_t *record = *it;
        BOOST_MESSAGE("Timestamp: " +
                boost::posix_time::to_simple_string(record->first) +
                "\tValue: " + boost::lexical_cast<string > (record->second));

    }
    BOOST_CHECK_EQUAL(result->get_count(), 1);
}
```

10.2 Module's output under regular run

```
[INFO 2010-Jul-04 13:03:34.752227] [poolmanager.cpp:26:poolmanager] Initializing
thresholds
[INFO 2010-Jul-04 13:03:34.752467] [poolmanager.cpp:28:poolmanager] Initializing
specifications
[DEBUG 2010-Jul-04 13:03:34.752590] [poolmanager.cpp:268:init_time_intervals_]
Found interval 'i1': 2010-Jul-04 11:23:34 <--> 2010-Jul-04 13:03:34
[DEBUG 2010-Jul-04 13:03:34.752655] [poolmanager.cpp:268:init_time_intervals_]
Found interval 'i2': 2010-Jul-04 12:53:34 <--> 2010-Jul-04 13:03:34
[DEBUG 2010-Jul-04 13:03:34.752716] [poolmanager.cpp:268:init_time_intervals_]
Found interval 'i3': 2010-Jul-04 12:53:34 <--> 2010-Jul-04 12:58:34
[DEBUG 2010-Jul-04 13:03:34.752813] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'c1'(l: 63; u: 65; w: 0.10000000000000001)
[DEBUG 2010-Jul-04 13:03:34.752871] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'free_c1'(l: 0.69999999999999996; u: 1; w:
0.59999999999999998)
[DEBUG 2010-Jul-04 13:03:34.752929] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'free_c2'(l: 0.40000000000000002; u: 0.68999999999999995;
w: 0.34999999999999998)
[DEBUG 2010-Jul-04 13:03:34.752986] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'free_c3'(l: 0; u: 0.39000000000000001; w:
0.050000000000000003)
```

```
[DEBUG 2010-Jul-04 13:03:34.753042] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'rws_c1'(l: 0; u: 0.01; w: 0.69999999999999996)
[DEBUG 2010-Jul-04 13:03:34.753101] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'rws_c2'(l: 0.01; u: 2.9900000000000002; w:
0.20000000000000001)
[DEBUG 2010-Jul-04 13:03:34.753158] [poolmanager.cpp:301:init_conditions_] Found
condition: Condition 'rws_c3'(l: 3; u: 99999.899999999994; w:
0.10000000000000001)
[DEBUG 2010-Jul-04 13:03:34.753251] [poolmanager.cpp:354:init_factors_] Found
factor (free_space): Factor 'free'(w: 0.59999999999999998; repo_type: 1;
interval: 2010-Jul-04 11:23:34 <--> 2010-Jul-04 13:03:34; no conditions: 2)
[DEBUG 2010-Jul-04 13:03:34.753337] [poolmanager.cpp:354:init_factors_] Found
factor (reads_per_time): Factor 'blob_id'(w: 0.59999999999999998; repo_type: 0;
interval: 2010-Jul-04 11:23:34 <--> 2010-Jul-04 13:03:34; no conditions: 3)
[DEBUG 2010-Jul-04 13:03:34.753418] [poolmanager.cpp:354:init_factors_] Found
factor (writes_per_time): Factor 'blob_id'(w: 0.40000000000000002; repo_type: 0;
interval: 2010-Jul-04 11:23:34 <--> 2010-Jul-04 13:03:34; no conditions: 3)
[DEBUG 2010-Jul-04 13:03:34.753481] [poolmanager.cpp:380:init_scenarios_] Found
scenario 's1': ( target: [EVERY]; total weight: 1; factors: blob_id blob_id )
[INFO 2010-Jul-04 13:03:34.753520] [poolmanager.cpp:30:poolmanager] Initializing
misc options
[INFO 2010-Jul-04 13:03:34.753565] [poolmanager.cpp:67:run] Starting cycle 1/1
[INFO 2010-Jul-04 13:03:34.753602] [poolmanager.cpp:68:run] Getting active
providers
[INFO 2010-Jul-04 13:03:34.755020] [poolmanager.cpp:77:run] Analyzing pool
[DEBUG 2010-Jul-04 13:03:34.816245]
[poolmanager.cpp:120:get_critical_providers_] Computed critical providers to:
[INFO 2010-Jul-04 13:03:34.816339] [poolmanager.cpp:139:action_] Analyzing
target: (192.168.212.128)[EVERY]
[INFO 2010-Jul-04 13:03:34.816377] [poolmanager.cpp:140:action_] Removing
critical providers from the list
[DEBUG 2010-Jul-04 13:03:34.880498] [implementation.cpp:89:is_true] CPS: 0(0,
6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:35.175611] [implementation.cpp:89:is_true] CPS:
0.3705(2223, 6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:35.175939] [implementation.cpp:89:is_true] CPS:
0.3705(2223, 6000, 0.01, 2.99 )
[DEBUG 2010-Jul-04 13:03:35.177179] [poolmanager.cpp:159:action_] A score of 0.5
has been computed for target (192.168.212.128)[EVERY]
[INFO 2010-Jul-04 13:03:35.177427] [poolmanager.cpp:168:action_] Successful
operation.
[DEBUG 2010-Jul-04 13:03:35.235982]
[poolmanager.cpp:120:get_critical_providers_] Computed critical providers to:
[INFO 2010-Jul-04 13:03:35.236314] [poolmanager.cpp:139:action_] Analyzing
target: (192.168.212.130)[EVERY]
[INFO 2010-Jul-04 13:03:35.236377] [poolmanager.cpp:140:action_] Removing
critical providers from the list
[DEBUG 2010-Jul-04 13:03:35.297808] [implementation.cpp:89:is_true] CPS: 0(0,
6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:35.597774] [implementation.cpp:89:is_true] CPS:
0.464167(2785, 6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:35.598075] [implementation.cpp:89:is_true] CPS:
0.464167(2785, 6000, 0.01, 2.99 )
[DEBUG 2010-Jul-04 13:03:35.598579] [poolmanager.cpp:159:action_] A score of 0.5
has been computed for target (192.168.212.130)[EVERY]
[INFO 2010-Jul-04 13:03:35.598684] [poolmanager.cpp:168:action_] Successful
operation.
[DEBUG 2010-Jul-04 13:03:35.862984]
[poolmanager.cpp:120:get_critical_providers_] Computed critical providers to:
[INFO 2010-Jul-04 13:03:35.863120] [poolmanager.cpp:139:action_] Analyzing
```

```
target: (192.168.212.131)[EVERY]
[INFO 2010-Jul-04 13:03:35.863194] [poolmanager.cpp:140:action_] Removing
critical providers from the list
[DEBUG 2010-Jul-04 13:03:35.932974] [implementation.cpp:89:is_true] CPS: 0(0,
6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:36.203550] [implementation.cpp:89:is_true] CPS: 0(0,
6000, 0, 0.01 )
[DEBUG 2010-Jul-04 13:03:36.203672] [poolmanager.cpp:159:action_] A score of
0.69999999999999996 has been computed for target (192.168.212.131)[EVERY]
[INFO 2010-Jul-04 13:03:36.203734] [poolmanager.cpp:164:action_] Removing
providers @ (192.168.212.131)[EVERY]
[DEBUG 2010-Jul-04 13:03:36.203792] [mover.cpp:39:stop] Running command
scripts/stop.sh 192.168.212.131
[INFO 2010-Jul-04 13:03:36.367137] [poolmanager.cpp:168:action_] Successful
operation.
[INFO 2010-Jul-04 13:03:36.367259] [poolmanager.cpp:105:run] Job done.
[INFO 2010-Jul-04 13:03:36.367297] [poolmanager.cpp:110:run] Exiting...
```