“ POLITEHNICA” UNIVERSITY OF BUCHAREST

FACULTY OF AUTOMATICS AND COMPUTER SCIENCE

Building an interface to access IAAS for BlobSeer

Supervisors

Prof. Dr. Ing. Valentin Cristea

As. Drd. Eliana Tîrşa

Student

Dorneanu Daniela

BUCHAREST 2010

Abstract

The Cloud model opens a new perspective in computing. Software and hardware resources are provided on demand and “as-a-service”; the user only needs an Internet connection to access the amount of computing power his project requires.

The main objective of this bachelor's project is to integrate BlobSeer distributed storage application with the Nimbus Cloud, and make it accessible as a storage service on the Cloud. To achieve this goal we set up a laboratory Nimbus environment, installed BlobSeer entities on virtual machines deployed into the Cloud and implemented a new feature that allows BlobSeer to restart from a consistent state, using a model of incremental checkpoints. We made a series of tests to validate the implemented solution. This report also presents the architectural model of the Cloud - as implemented by Nimbus and our solutions to many undocumented Nimbus' new version configuration problems.

Keywords: Virtualization, Nimbus, GridFTP, Service Provider, Storage System, Blob, Metadata, Security.

Table of Contents

1 Project objectives

The goal of this bachelor's project is integrating the BlobSeer distributed storage system into the Nimbus Cloud environment. To achieve this, we have implemented the following objectives:

- Set up a laboratory Cloud environment using the Nimbus, as described on chapters *4.1.1 Nimbus Deployment and 4.1.2 Virtualization Solution – Xen Setup*
- Deploy BlobSeer entities on various virtual machines in the Cloud, as described in chapter *4.1.3 BlobSeer Setup*
- Add checkpoint-restart functionality to BlobSeer, in order to enable scheduled restarts, as described in chapter *4.2 BlobSeer checkpoint-restart inside Nimbus Cloud.*
- Provide an interface for access and control BlobSeer in the Cloud system as described in chapter *4.3 Command Line Interface for controlling BlobSeer in the Cloud.*
- Perform various tests to validate the implemented solution, as described in chapter *4.4 Tests*

2 Introduction and motivation

The way the Internet has evolved in being very easy to access has opened new perspectives regarding how this facility can be used in computing. Computational power has become available through models like Clusters, Grids and nowadays a new model has more and more impact on the way we do computation – the Cloud infrastructure. The cloud-computing concept defines an abstraction over a network of computers, hiding the physical infrastructure from the user and providing (often virtualized) resources as services. Each service consumer has access to the cloud from its own desktop, notebook, PDA by simply using an Internet connection.

This report presents a complete setup of a Cloud using Nimbus [23] and an integration of the BlobSeer[25] distributed file system into this Cloud, using Xen [24] as a virtualization environment. We implemented mechanisms for starting/stopping/restarting BlobSeer inside the Nimbus Cloud , while preserving the data it stored during previous runs and bringing BlobSeer to a consistent state before stopping it. We also provide a command-line interface for these actions. This report also presents the architectural model of the Cloud - as implemented by Nimbus and our solutions to many undocumented Nimbus' new version configuration problems. We will also state the advantages of using BlobSeer as a storage system in a IaaS Cloud [1].

The type of Cloud we have experimented in this project is Infrastructure-as-a-service (IaaS). This model represents a fundamental change of perspective: "when a remote user "leases" a resource, the control of that resource is turned over to that user".[9] Renting configurable resources was enabled by the availability of free and efficient virtualization technologies like the Xen hypervisor. We used an open-source implementation of IaaS Cloud, Nimbus , which is in a rapidly-growing development process and even reached a stage when it can be used in an Enterprise Environment, as a good alternative for well-known commercial solutions (e.g. Amazon EC2 [13]).

Making the IaaS model available for wide usage raises many new challenges like security measures, failure resistance, huge amounts of data and interoperability. When working with huge amount of data and also with huge files, which is a common situation in a Cloud system, manageability can become an issue. Any problem that occurs can lead to very big losses of data.

BlobSeer is a data storage application specially designed for distributed systems that was built as an alternative solution for the existing storage layers currently developed by Amazon [14], Google[6] or the open-source storage system Hbase[26], used by Hadoop[27]. By keeping the assets of the previously mentioned storage services: decentralization, interface for data-location-awareness, fine-grained data access, and by adding new design principles like versioning, BlobSeer has a considerably improved speed when comparing with other similar systems.

A very important subject addressed in this report is preparing BlobSeer for the Cloud environment. Because BlobSeer is still in the state of research this system has some shortcomings. One of them, that needs to be resolved before including BlobSeer in the Cloud system is not having the base to start from a consistent state; it always starts from zero data.

Adding support to start BlobSeer from a consistent state previously saved is a significant step. This aspect is very important in the Cloud context as the user needs to be able to shutdown the application on the Cloud when it is not used and restart it from the last consistent state. This is also an important feature for the failure tolerance of the system.

A key feature of BlobSeer is being able to do most of its operations in parallel using locking as little as possible. This is achieved by the versioning system which assures the consistency of the written data. The single operation which is carried out in a serial fashion is updating the version of the data recently written. We studied various ways of saving the state of the system without affecting the almost lock-free implementation of BlobSeer and still having proper snapshots of the application. The conclusion was developing support for taking uncoordinated snapshots – this meaning that each process will store each consistent state. Main reasons for choosing this solution are overall good performance and the autonomy of each entity to save its state in the most convenient moment. This choice also maintains the good performance BlobSeer has without our module and also made it an application well integrated with the Cloud. In this report we will describe in detail this approach of implementing checkpoints for BlobSeer and we will also develop proper tests to validate the chosen implementation.

This report starts with a presentation of the related work in the field of our project. The *3Background* chapter starts with the Cloud computing technologies in general, describing in particular the cloud storage implementations for the cloud, and continues with presenting the specific applications used in this project the implementation of the IaaS Cloud Nimbus and the storage layer BlobSeer. The background chapter also describes the various checkpoint-restart approaches that can be used for a distributed storage application. The part *4 Our Contribution* contains the implementation of this project consisting in a detailed setup description, our method to add storage consistency for BlobSeer, the interface that provides access to BlobSeer in the Cloud and the tests to verify our implementation. The chapter *5 Resources* specifies the hardware and software base we used for this project. In the sixth chapter, *6 Conclusions and future developments* we emphasize our final opinion and possible directions of development for this project.

3 Background

3.1 Cloud computing overview

3.1.1 Cloud computing in the context of distributed systems

Cloud computing is a new paradigm shift in computing which implies that shared resources and applications are provided to Desktops or other end-user devices on-demand through the Internet. With Cloud system computation becomes an abstraction built upon clusters of servers, an abstraction transparent to the client. Cloud computing implies delivering services through the Internet in a dynamically scalable way and typically using virtualization. [7]

There are two key elements that advantage clouds in respect to other computing models from the point of view of resource usage. The first limitation starting from supercomputers and single servers was the static and finite amount of resources that could be shared – With this type of technologies new resources would usually be acquired when the current ones won't cover the needs any more. But adding new resources to the network requires a lot of work to be done by the network administrators and thus time to get the best solution on short and long term. This operation however can prove to be redundant on long term – in most of the cases more resources are required only for short periods of time before a project's deadline, during load testing. [2] Being able to approximate what resources might be necessary in the future it's very difficult and the usual politic for this is to get new resources when the systems crashes – which usually leads to loss of data and frustration for developers. Besides some busy periods the resources are idle so the investment wasn't very efficient. This is one of the reasons why getting resources on the network only on demand can be a very good idea to efficiently use resources only when they are needed. [1]

Grid computing is a model which allows including new remote resources in the network. This system is based on Virtual Organizations in which volunteers get to participate in a certain project by putting together computational power – their own clusters. [5] The main advantage of grid computing is being able to easily add new nodes into the Grid and use them only when they are needed. This technology allows the scientific and industrial environment to collaborate into developing big projects which requires a high computing power. Each node will receive tasks from the job manager, will compute data independently and then will send the results back to a special node. New nodes can be added easily in the network by connecting them to the Grid. A weak point of Grid computing when compared to supercomputers and dedicated clusters – is on the reliability side, as it could be a usual thing that a remote computer from the Grid loses connectivity, in which case the task could be lost and thus redistributed to a different node. This lost task can lead to losing other data that depend on this node's results. The nodes in the Grid should respect a certain protocol and the Grid users need to have a valid certificate and to respect certain specification used in the Grid. However the infrastructure of grid computing has some limitation based on the fundamental assumption of Grid computing: control over the mode in which remote resources are provided belongs to the remote site. Grid users usually don't have administration rights on the remote resources for security reasons – and this will limit the ability to deploy software that need root access to the resources. Also occasionally the users have to adapt their application in order to run using the software offered by the Grid. Besides that Grid doesn't offer a way for the users to receive on demand computational power. In general, a user can use the Grid if is a member of a virtual organization that was approved in the Grid. [2]

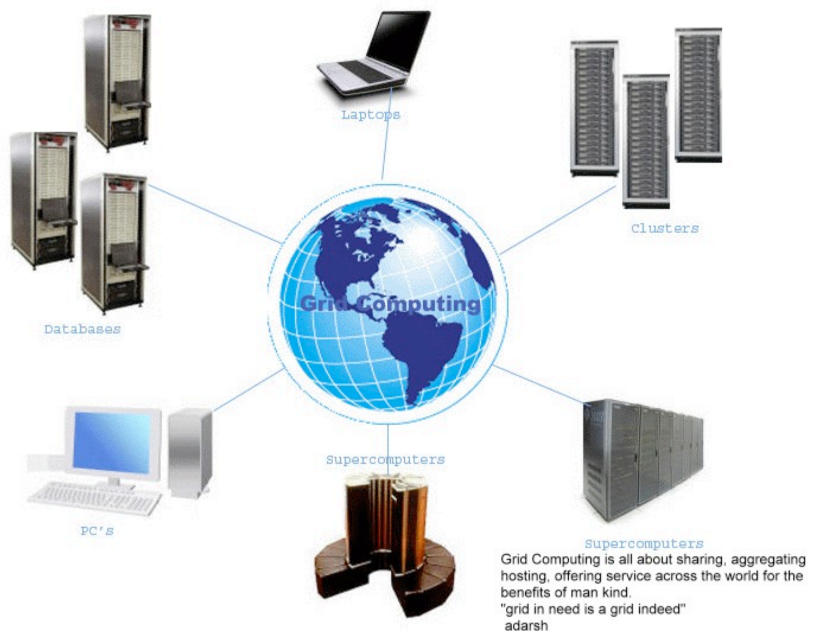
Laptops
Clusters
Grid Computing
Databases
Supercomputers
PC's
Supercomputers
Grid Computing is all about sharing, aggregating hosting, offering service across the world for the benefits of man kind.
"grid in need is a grid indeed"
adarsh

Fig 1 – The devices that can be part of the Grid

The cloud-computing concept defines an abstraction over a network of computers which allows the cloud user to see the entire network as a single PC. Each service consumer – user – has access to the cloud from its own desktop, laptop, PDA using an Internet connection. The hardware and the operating systems that operate on this hardware is invisible to the client. Each request passes through the system management which finds the correct resource and then calls the system's services necessary to fulfill the request. These services access the needed resources from the cloud and launch the appropriate web applications or creates/opens a certain document, or gives remote access to a virtual-machine setup by the user. Afterwards the system's monitoring and metering functions track the usage of the cloud. [1]

The paradigm of cloud computing made it possible the access existing resources at remote sites when these are needed. The cloud is able to deliver more resources and on demand offer them on demand to the user. The client has complete access to it's resources in the cloud and he can make his own setup. [2]

The Cloud computing concept is based a lot on virtualization. Therefore in the cloud the client has the opportunity to control the software stack from the operating system to the application level. This specialized type of cloud is known as IaaS (Infrastructure-as-a-Service) – as it offers an entire infrastructure to the users. IaaS is currently provided by Amazon, GoGrid and Nimbus. There are other types of cloud such as Platform-as-a-Service (what Google and Microsoft are offering) or Software-as-a-service (community specific applications and portals).

The key points that make cloud-computing a very good and attractive solution for dynamically control the amount of used resources are the user's ability to have complete administration rights over its resources, the customizable environment and the possibility to use those resources on-demand. The Client sees only his resources while the cloud infrastructure is transparent for him. The users only needs to care about their own list nodes into the cloud. [1]

One of most obvious difference between Cloud computing and Grid computing is the fact that Cloud offers more control to each individual user of the cloud, no matter the size of his job while Grid computing is specialized to support large set of users organized in virtual organizations. In order to best benefit from the data-intensive storage within the Grid computing technology the amount of distributed data must be large, while storing data into the Cloud has to be economically profitable even for small amounts of data. [3] In other words the Cloud seen as IaaS represents a fundamental change of perspective: “when a remote user “leases” a resource, the control of that resource is turned over to that user”. This change in assumption was enabled by the availability of a free and efficient virtualization technology: the Xen hypervisor. Before virtualization, turning over the control to the user was fraught with danger: the user could easily subvert a site. But virtualization provides a way of isolating the leased resource from a site in a secure way that mitigates this danger. Virtual machines can be deployed very fast (on the order of milliseconds) when, in addition to that, the overhead and the price associated with a reliable virtualization technology went down, it suddenly became viable and cost-effective to use them in order to lease resources to remote users. [7]

This new technology is supposed to change the way the computers are seen today – the way we run applications and store information. The individual computer is changed with the nebulous collection of all types of computers which are accessed through the Internet. One big plus from the business point of view is the fact that companies can use the Cloud to scale up massive capacities without worrying any more about the cost of the infrastructure removing from their businesses other additional expenses like the entire hardware infrastructure, the room for keeping these servers, hiring and training new personnel for maintaining the environment, licenses of new software or even additional costs for growing the infrastructure – which will be used rarely. With cloud computing it is certain that one will pay only for the used resources.[1] When using the Cloud the service consumer becomes independent of the PC or of the specific version of it's device. The user is not responsible for the infrastructure he uses and he is not aware of where that infrastructure is located as the entire Cloud is hidden under a single domain, which exposes all its services.

Being able to elastically enlarge and diminish the effectively used resource enhances new challenges we have previously reviewed. This is why nowadays there are a lot of big companies like Google and Amazon that already offer Cloud Computing services and are investing a lot in developing this technologies and offer much more reliability, trust and lower cost for “renting a byte” in the Cloud. On the market there are also some very good open source alternatives for Cloud toolkits like the Nimbus Toolkit. We use Nimbus to setup the cloud environment for this project.

3.1.2 Understanding cloud services

Before start working with a cloud framework it's important to understand the main advantages of cloud-computing in the distributed systems environment and how the components interact into the cloud.

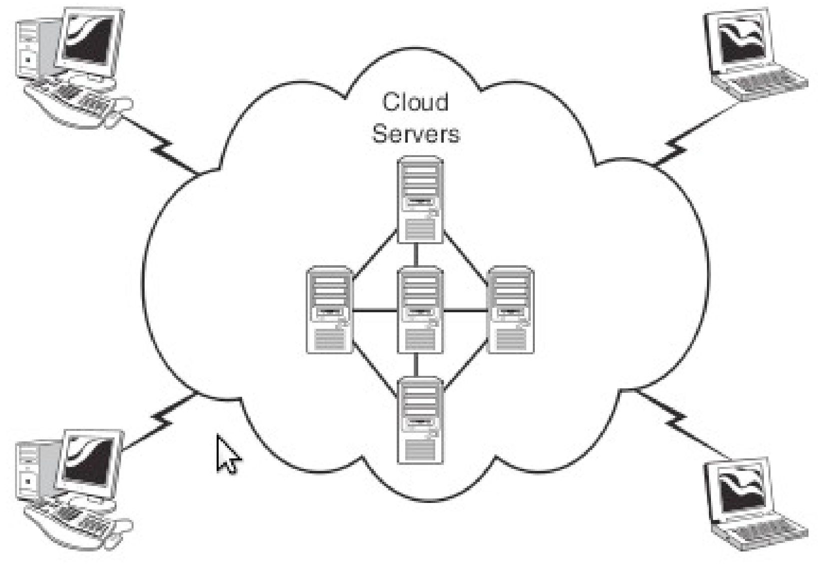

Fig 2 How users connect to the cloud

As represented in the Fig 2 [2] each service consumer has access to the cloud from its own desktop, laptop, PDA using an Internet connection. For these users the cloud is seen as a singular entity (application, device, document, OS of a virtual machine). The hardware and the operating systems that operate on this hardware is invisible to the client. Each request passes through the system management which finds the correct resource and then calls the system's services necessary to fulfill the request. These services access the needed resources from the cloud and launch the appropriate web applications or creates/open a certain document, or gives remote access to a virtual-machine setup by the user. The monitoring and metering functions track the usage of the cloud.

All the web based applications or resources offered via cloud computing are considered Cloud services. For examples the Google/Docs application, Google Calendar, Microsoft Live are hosted on the cloud – and the user is able to access these services usually through the browser. After launching the application it behaves as an usual Desktop application.

The advantages of cloud services are numerous: if the system crashes, this doesn't affect the document at all, the document can be accessed from different locations and also it can be accessed and modified from more than one user simultaneously.

Types of Cloud Services

- Software-as-a-Service (SaaS)

This type of service offered in the Cloud is the most common of the services that the Cloud can provide. This service provides an application on it's own central server that can be accessed by thousands of users without them installing any new application through Internet, Intranet, LAN, or VPN. Key words regarding this type of cloud are: Internet connection, centralized control, one-to-many model. Examples of this type of application: Google Docs, Google Apps, Microsoft Live.

– Platform-as-a-Service (PaaS)

This service provides a computing platform or a computing stack as a service, this meaning consuming cloud infrastructure and sustaining cloud applications. This type of cloud computing service eliminates the need for a lot of companies to buy their own hardware for development and testing and even for deploying their applications. Besides offering computational power and storage PaaS also offers an operating system which is patched and kept up to date by the cloud service provider. Some examples of this type of application are: Microsoft's Windows Azure Platform (an operating system which serves as a runtime environment for a set of services and also provides a platform for development), Google App Engine (an application for developing and hosting web applications in Google-managed data centers). PaaS is the best approach if customization is not an important asset as the remote resources are already set up with a default set of applications. [12]

– Infrastructure-as-a-Service (IaaS)

IaaS Clouds deliver on-demand computing power and storage. With IaaS users can deploy an image that includes applications and update the components of that image after the user's needs. An example of a IaaS implementation is Amazon Web Services (Amazon EC2 and Amazon S3). With IaaS the enterprise can make their own virtual machines or can ask the Cloud administrator to prepare the machine with certain requirements. Customers select and basic software servers for their part of the cloud and then load up their libraries, applications and data then configure them themselves. Virtualization enables IaaS providers to offer almost unlimited instances of servers to customers and make cost-effective use of the hosting hardware. BlobSeer was built as an application for distributed environments as its main focus is offering features to work very fast with processed data that is written in parallel on different nodes of the cluster. Therefore there are a lot of advantages offered by BlobSeer once it is integrated on the Cloud. [7]

3.1.3 Cloud storage

One of the most important features of the Cloud is being able to store huge amount of data. Keeping data into the cloud is more gainful then saving it on dedicated servers and it's also more secure as we are protected from accidentally deleting the data as it is saved on multiple machines. When saving files on the cloud the user sees a virtual server which contains it's data as being saved as on a static machine having a certain name. However behind this interface the file is usually divided between many third-party servers (not dedicated ones) – and the data location changes continuously into the cloud. All these features involve having a special storage layer that implements these special requirement. The main initiatives regarding solutions for providing storage on Cloud system are

GoogleFS, Amazon S3 and the storage layer for Hadoop. Next we will shortly present the main characteristics of these applications.

The main features offered by the GoogleFS are the following: [4],[6]

- Splitting the huge files into 64 MB fixed blocks – distributed among chunk servers. These chunks are extremely rare overwritten. These chunks are usually read or appended to these big chunks of data. The nodes of the cluster are divided into two types one Master node and a lot of Chunkservers.
- The metadata that describes the directory structure of the file system and the metadata that describe the chunk layout are stored on the centralized Master server. The Master server receives updates periodically from each chunk server ("*Heart-beat messages*").
- The system that provides the permission to modify a chunk of data works through “expiring leases” of time. During this time granted by the Master no other process will be granted permission to modify that chunk of data. The modified chunkserver, which is always the primary chunk holder, then propagates the changes to the chunkservers with the backup copies. The changes are not saved until all chunkservers acknowledge, thus guaranteeing the completion and atomicity of the operation.
- Programs access the chunks by first querying the Master server for the locations of the desired chunks; if the chunks are not being operated on (if there are no outstanding leases), the Master replies with the locations, and the program then contacts and receives the data from the chunkserver directly.

In HDFS, the read operation essentially works the same way as with GoogleFS. However this is different in semantics: it allows only one writer at a time and once written the data cannot be altered by appending or overwriting. HDFS also has some features that data throughput similar to the one implemented by GoogleFS.

With Cloud-computing development the need for a system that offers effective data-intensive application increased. Thus Amazon started to offer Map-Reduce infrastructure as a service – Elastic Map-Reduce. Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. [14] Hadoop provides support for using as a filesystem the Amazon S3 (Amazon Simple Storage Service) instead of HDFS through S3 Native FileSystem (URI scheme: s3n) or the S3 Block FileSystem (URI scheme: s3). The main design principles of the S3 file-system are the following [13]:

- S3 System was built with the idea of simplicity in mind and the objects can reach sized up to GB. It implements simple and well established standards as SOAP. Each object is stored in a bucket and retrieved via a unique, developer-assigned key. The user is able to delete, read and write to this objects – identifying them by this unique Id.
- Data-location-awareness of the data is another characteristics of S3 file system.

Objects can be created to belong to one or more regions. Once stored in a Region the block can never leave the Region unless the user transfers it out. For example, objects stored in the EU (Ireland) Region never leave the EU.

- The “decentralization” principles has the target to remove scaling bottlenecks and single points of failure.
- By “tolerance to failure” the system continues with minimal interruption even if some components failed.
- The S3 file-system has a “controlled parallelism”, which implies implementing the necessary granularity to improve performance
- The “symmetry” characteristic means that all nodes are identical as functionality and don't require node specific configuration and “simplicity”.

These principles make the S3 system to be very efficient and easy to grow.

Other initiatives are:

- GluserFS – a general purpose distributed file systems – which works with storage bricks over the network. [15]
- General Parallel File System (GPFS) is a high-performance shared-disk clustered file system developed by IBM. It is used by many of the supercomputers that populate the Top 500 List of the most powerful supercomputers on the planet. [16]
- The Parallel Virtual File System (PVFS) is an Open Source parallel file system. A parallel file system is a type of distributed file system that distributes file data across multiple servers and provides concurrent access by multiple tasks of a parallel application. PVFS was designed for use in large scale cluster computing. PVFS focuses on high performance access to large data sets. It consists of a server process and a client library, both of which are written entirely of user-level code. A Linux kernel module and PVFS - client process allows the file system to be mounted and used with standard utilities. The client library provides for high performance access via the message passing interface (MPI). [17]

In this project we have chosen to integrate BlobSeer, an open-source alternative to the systems we have presented in this chapter – GoogleFS, Amazon S3, GPFS, PVFS – into the IaaS Cloud system Nimbus.

3.2 Nimbus

3.2.1 Overview

Nimbus is an open-source framework that can be used to enhance a cluster to be used as IaaS. The goal of the project is to develop a Cloud system mainly for research purposes but can be used not only for research. Through this system the client is able to leverage existing resources at remotes sites by deploying virtual machines on the remote servers and configure them according to the users' needs. The difficulty of deploying Nimbus can vary from very simple to complex depending of the Cloud's size or of scope. However for a cloud with a lot of nodes Nimbus offers a service named “context broker” which allows clients to install and coordinate large virtual clusters automatically in parallel. In the current project we have used a light setup of a cloud service with the minimum number of elements. The principal elements of the Nimbus system are: the Virtual Machine Manager, the Repository, the Workspace-service, the Client.

3.2.2 Nimbus Architecture

We will shortly describe here each part of the Cloud and what we have installed on each element – All these constituent parts are able to communicate with each other as represented in the Fig 3 [12]. Security between the components of the cloud is addressed using X.509 cryptography standard and the certificates are generated as described in [18].

A standard configuration model that we have worked with is represented in Fig 2.3 [12] and contains the following components [23]:

– The Workspace-Service

The Workspace service is the component that publishes information about each Workspace-Constrol. The Workspace service is a standalone site which can be invoked by various remote protocols. Here we have installed the nimbus service, version 2.4 from official nimbus project download page. The Workspace Service site manager is the instance to where the client is sending the commands to be executed on the cloud. In this setup the Workspace Service will copy the virtual machines from the Repository to the VMMs and deploy them. The deployed virtual-machines contain BlobSeer installed on them.

– The Repository

On this machine are stored the Xen Virtual-machines which will be deployed into the cloud. Here we installed globus-gridftp-server following the tutorial from the Globus website [18]. We used Gridftp – as it is an extension of the standard FTP which solves the problem of incompatibility between the storage and the access systems. GridFTP provides a uniform way of accessing the data, encompassing functions from all the different modes of access, building on and extending the universally accepted FTP standard. FTP was chosen as a basis for it because of its widespread use, and because it has a well defined architecture for extensions to the protocol (which may be dynamically discovered).

– Virtual Machine Manager

The VMM node is the node with which manages all the virtual machines. The privileged account on the service node needs to be able to SSH freely to the VMM (without needing a password). And vice versa, the privileged account on the VMM nodes needs to be able to freely SSH back to the Workspace Service nodes to deliver notifications. On this node we have installed the Workspace-Control and the Xen Hypervisor.

- ✓ The Workspace-Control is implemented in Python in order to be portable and easy to install. Requires libvirt, sudo, ebtables, and a DHCP server library. This component of Nimbus is installed on each VMM node in order to offer the next functionalities
  - start/ stop/pause the VMs;
  - reconstruct/modify/manage the VM images;
  - securely connect the VMs to the network (uses DHCP to assign Ips);
  - deliver contextualization information (using the context broker).
- ✓ Xen Hypervisor which allows several guest operating systems to execute on the same computer hardware concurrently. We used Xen technology to the detriment of KVM which also works good with Nimbus because Xen provides better interoperability, support for batch systems and works very well on hardware that doesn't have special support for virtualization. [24] To communicate with the Xen service the 2.4 version of Nimbus is using libvirt to manage the deployed virtual machines.

– The Client

The client has the goal to help the Cloud users to start using the IaaS system very easy from any sort of device – laptop, PDA and for that it offers an interface for communication with the Workspace-Service and the Repository. This component sends requests to the VMMs and the Workspace Service nodes. [23]

On this node we have installed the nimbus autocontainer. This tool has the goal to offer an easy support for the administration of the machines on the Cloud because: Generates full configuration strings from shortcuts, can transfer files to/from the repository node using its embedded GridFTP client, packages almost all needed dependencies and other useful utility programs.

The requests sent to the Repository are list (which will list all the available virtual-machines from the Repository), transfer (this will transfer the virtual machines). The requests sent to the Workspace-Service are run (which will send a virtual-machine from the Repository to a VMM and start it), terminate (shut-downs the virtual-machine). The run command offers very important information for the administrator of the system: like the IP of the new deployed machine. This information is used when launching BlobSeer instances into the Cloud after the Virtual Machines are already started. Requesting certain fixed IP into the cloud costs much more than having allocated random IPs.

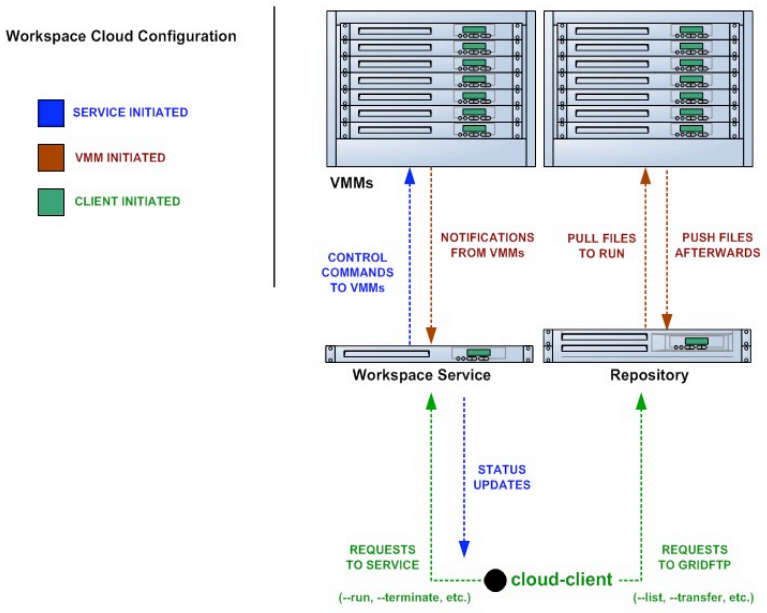

Fig 3 Overview of the cloud configuration

3.3 BlobSeer

3.3.1 BlobSeer Overview

The BlobSeer data storage layer is a software that was built as a more efficient alternative than the HDFS the default storage layer for Hadoop, by eliminating some of it's disadvantages. When working with huge amount of data and also with huge files the problem that arises it's the manageability problem – any problem that occurs can lead to very big losses of data. The storage layer proposed by BlobSeer supports versioning – a feature which allows rolling back undesired changes, but also dividing a dataset into independent datasets that can evolve by themselves. Another plus that BlobSeer adds it's an interface that permits the application to be data-location-aware – therefore the scheduler is able to place computational work close to the data, reducing the network traffic. Providing a high throughput in spite of heavy concurrent access is another feature that the storage layer provides by eliminating the locks as much as possible and making all the activity to take place in parallel. This design of this application was inspired from the previous similar storage layers like HDFS, GoogleFS, S3 from Amazon – and it is faster then these applications because it eliminates some of those weak points.

BlobSeer is an application specially designed to deal with the requirements of large-scale data-intensive distributed applications. Data is abstracted as large sequence of bytes and stored as a blob (binary large object), which is essentially a huge file. [25] BlobSeer took act of all these previously developed systems' lacks and pluses which were used as a start point to design the BlobSeer as a concurrency-optimized file system for Hadoop. The BSFS layer enables Hadoop to use BlobSeer as a storage backend through a file system interface. [4]

BlobSeer design advantages:

- the system which is based on versioning the objects offers a lock free access to data and also favors scalability under heavy concurrency.
- High throughput it's assured by the decentralized data and metadata management.
- The scheduler is able to place computational work close to the data, reducing the network traffic.

In BlobSeer the data is structured into flat sequences of bytes – also called BLOBs (Binary Large Objects). The size of the blob can be very small (64mb) for fine-grained access to huge BLOBs of hundred of GB.

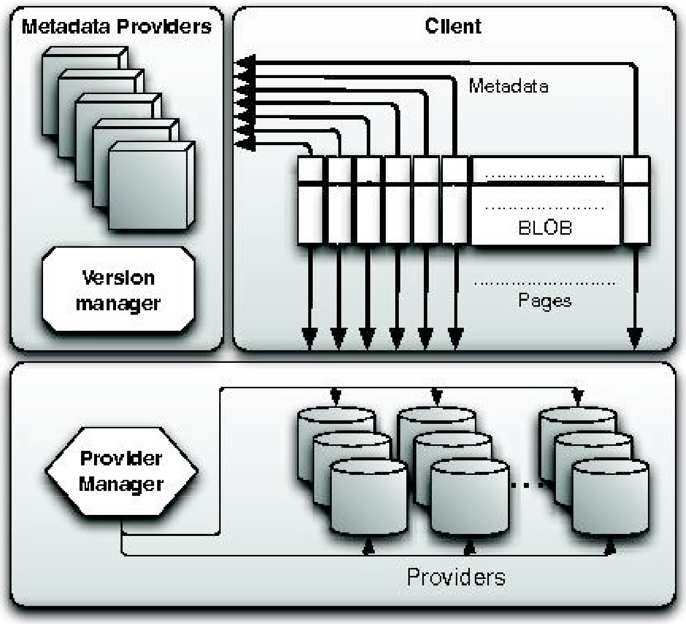

Fig 5 BlobSeer's Architecture

BlobSeer represents a set of distributed communication processes – the distribution and interaction between them are shown in the Fig 5 [25] From the Blobseer design we can extract the following entities [11]:

1) Client

This role is the one the can *create/write/read* from BLOBs. All these operations are done concurrently even there are done with the same BLOB.

2) Data provider

The data provider or simply the provider is the item that stores the data generated by the writers. The new data providers can dynamically join or leave the system. In the context of Hadoop Map-Reduce, the nodes hosting data can also act as computing elements as well. This helps the node to benefit from the Hadoop strategy to make data aware of it's location, thus placing the computation as close as possible to the data.

3) Provider Manager

The provider manager maintains information about available storage space and schedules the placement of newly generated blobs. This entity selects using a load-balancing algorithm the providers to which it can evenly assign the data.

4) Metadata Provider

This items item physically stores the metadata that allows identifying the blocks that form a snapshot version. The access to the metadata providers can be done concurrently. The nodes that host this type of items can also host computational power.

5) The version manager

This item assigns incremental snapshot versions numbers so that the serialization and atomicity of the write operation is assured. It is recommended to host this node on a separate server.

Currently the 2 most important operations read/write are the ones implemented on BlobSeer which will be shortly described as follows:

1) Reading [11]

When reading the client has to do the following operations which can be easily read from Fig 6:

- find out the BLOB that corresponds to the requested file;
- then the client must specify the version manager from where to read, and also the range (size + offset) that needs to be read. This data need to be provided considering the Hadoop doesn't support versioning yet the reading call is done in the current implementation.
- Afterwards the client asks the version manager about the requested version of the BLOB. The version manager forwards this request to the metadata providers, which in turn send on back to the client the metadata that correspond with request of the client – the range (size, offset) from where the client will read the data. One the location of the data was identified the client fetch the blocks from the data providers.
- All the requests are sent independently one another and can be processed in parallel by the data providers.

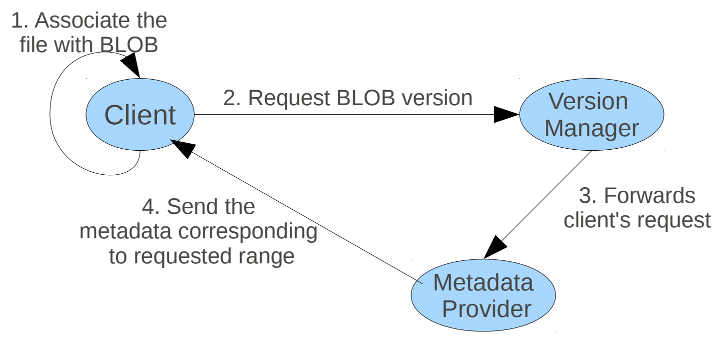
1. Associate the file with BLOB
Client
2. Request BLOB version
Version Manager
3. Forwards client's request
Metadata Provider
4. Send the metadata corresponding to requested range

Fig 6 BlobSeer reading scheme

2) Writing [11]

When writing the client has to do the following operations which can be easily read from Fig 7:

- When writing the first operation the client does is splitting the data to a list of block in order to correspond to the requested range.
- Then the client sends a request to the Provider Manager and asks it for a list of available Providers, where the writer would then write it's data: one provider/per block.
- The Provider Manager chooses the Providers following a load-balancing algorithm and sends the list back to the client.
- The client then will write the data in parallel on these Providers. If one writing fails then all writes will be considered as failed.
- If all the write operations are successfully accomplished the client contacts the VM to announce it of it's intent to update the BLOB's version. As described earlier this first phase it's done in parallel independent of other writers.
- Afterwards the VM assigns the each write request a new snapshot version manager id. If the VM Manager detects that lower versions of the data the client wants to update currently is still in the writing process then it gives the client a message concerning this. When all the lower versions of the current BLOB to be written by the client are ready the VM sends the id to the client.
- The client will receive this id and will generate new metadata that will send it together with the existing metadata to a Metadata Provider – that will store this metadata. This operations creates the illusion of a new stand alone snapshot.

- When the Metadata Provider successfully has written the data will notify the client about this.
- When the client receives the success notification from the Metadata Provider the Version Manager will be notified of the existence of the new version of the BLOB.

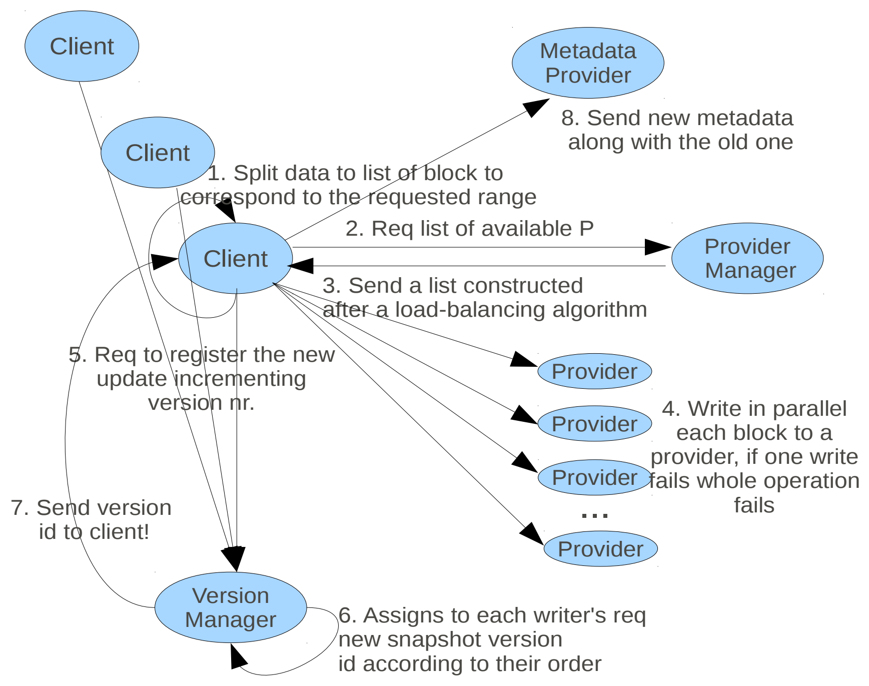
Client
Client
Client
Metadata Provider
8. Send new metadata along with the old one
1. Split data to list of block to correspond to the requested range
2. Req list of available P
Provider Manager
3. Send a list constructed after a load-balancing algorithm
5. Req to register the new update incrementing version nr.
Provider
Provider
Provider
...
Provider
4. Write in parallel each block to a provider, if one write fails whole operation fails
7. Send version id to client!
Version Manager
6. Assigns to each writer's req new snapshot version id according to their order

Fig 7 BlobSeer writing scheme

3.3.2 BlobSeer's advantages on the Cloud

BlobSeer is an application that serves as a storage layer when working with very big sets of data on distributed systems. Its strong points, which make BlobSeer a potential alternative for other similar storage layer for Cloud – GoogleFS, Amazon S3 – are the versioning system, achieving high throughput and the data-location-aware feature. We will present each of these advantages next. [4]

1) Versioning

A client from BlobSeer is able to create new blobs, write/append data to them, read data from them by using a simple interface. Each blob is identified by a unique ID in the system returned when the BLOB is created. The client also is responsible to version the BLOB because each time a subsequence of bytes it written to the BLOB a new snapshot reflecting the changes is generated instead of overwriting any existing data. The counter of the snapshot is auto-incremented -and all the past versions of the BLOB can be accessed – if they weren't removed to free space. When reading – by default the user is able to read from the most recent snapshot of the BLOB but also it is able to read from other versions of the blob through a special call. Every time a new version of the BLOB is created only the part that is different from the previous snapshot is actually saved. Also the new version of the BLOB shares all the unmodified data and most of the metadata with the previous BLOB. This design facilitates implementation of rollback and branching as the data and metadata corresponding to previous versions will remain this way easy to be accessed in the system. [4][11]

The BlobSeer as a storage layer for Hadoop aims to offer a high throughput under heavy access concurrency when reading, writing and appending data to the BLOB. The accomplish this goal the design uses data-striping, distributed-metadata, a design based on versioning, lock-free data access.

2) Data-striping

This technique implies that each BLOB containing blocks of fixed sizes can be distributed among the storage nodes. This strategy aims to distribute homogeneously the BLOB's blocks so that when the readers want to access concurrently different parts of a file the throughput to be maintained as high as possible.[4]

3) Distributed metadata

A BLOB is accessed by specifying a version number and a range of bytes delimited by an offset and a size. The metadata is organized as a distributed segment tree – which is associated with to each version of a given BLOB id. The node covers the range (offset, size). For each node that is not a leaf the left child covers the first half of the range and the second half the right child. To make more efficient the concurrent access to the metadata - the nodes of the tree are distributed to the *metadata-providers* using DHT (Distributed Hash Table). The expansion of the tree when creating a BLOB, overwriting or appending blocks from th e BLOB can be seen in the Fig 4 [4] Medata Representation. Each tree node it's identified in the DHT by the version number and by it's range (offset, size). The entire subtree are shared among the trees associated to the snapshot manager. The decentralized design of the metadata has a very important impact to maintaining a high-performance on concurrently accessing the BLOBs.

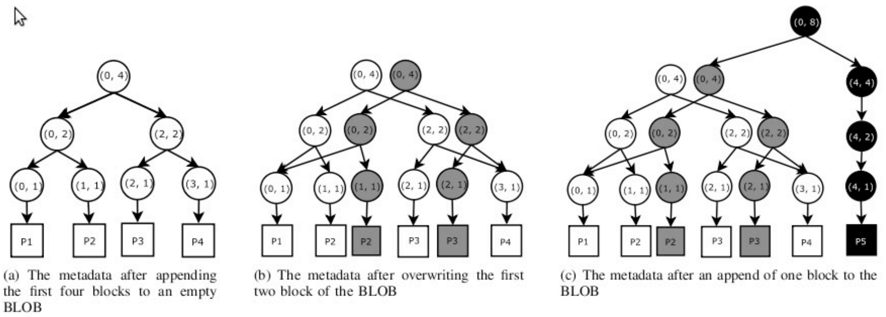
(0, 8)
(0, 4)
(4, 4)
(0, 2)
(2, 2)
(4, 2)
(0, 1)
(1, 1)
(2, 1)
(3, 1)
(4, 1)
P1
P2
P3
P4
P5
(a) The metadata after appending the first four blocks to an empty BLOB
(b) The metadata after overwriting the first two block of the BLOB
(c) The metadata after an append of one block to the BLOB

Fig 4 BlobSeer metadata representation

4) Versioning based concurrency control

The versioning based concurrency control algorithm relies on avoiding synchronizations both on data and metadata as no existing data or metadata is ever modified.

Considering that only the difference of the BLOB is stored each writer can send it's data independently of other writers to the corresponding data-provider. In the first phase the writing action can be done in parallel without any synchronization at all. In the second stage the version manager responds to the writers requests by assigning a version number then generate the corresponding metadata. The metadata of the new modified snapshot is sent together with the metadata of the lower versions simulating this way an independent snapshot. This second stage when the version manager responds to the writers assigning them a version number needs however to be serialized by the version manager. Afterwards the writers can function concurrently independent, thanks to the design of the metadata scheme. To assure the consistency of the read data the order in which new snapshots are made available for reading must be identical with the one assigned by the version manager. This is not an impediment for the concurrent writing.

Concerning the readers they are completely independent as they access permanent snapshots, the lower versions of the snapshots are always unmodified. Therefore the readers can function completely parallel.

This is why it can be affirmed that this design supports read/read, read/write, write/write in a concurrent way. This feature is a clear advantage of the BlobSeer over the HDFS – which doesn't support concurrent writes to the same file at all.

5) Strong consistency semantics

Readers are guaranteed to see a new write when the following two conditions are satisfied: (1) all metadata for that write was successfully committed and (2) for all writes that were assigned a lower version number, all metadata was successfully committed.

Since a writer is assigned a version number only after it has successfully written the data, condition (1) actually means: the writer has successfully written both data and metadata. Even though the write primitive may have successfully returned, the write operation as a whole may complete at a later time. Thus, condition (2) translates into: both data and metadata of lower version snapshots have been successfully written, so all previous snapshots are consistent and can be read safely. This naturally means the new snapshot can be itself revealed to the readers. Both conditions are necessary to enforce linearizability.

3.3.3 Developing on BlobSeer

BlobSeer is a file-system which has special features developed for distributed systems. It is built in C++ and uses API to work with Boost, Berkley DB, Libconfig libraries. A short presentation of these libraries is necessary in order to start developing on BlobSeer.

– Boost

Boost is a collection of free portable C++ source libraries. Boost libraries are intended to be widely useful and usable on a large spectrum of application that use C++. To find out more about Boost the Boost Official Website can be consulted [19].

– Libconfig

Libconfig is a simple library for processing structured configuration files which are written in a format more compact and more readable, which is also type aware if comparing with XML. [20]

– Berkley DB

Berkley DB is a transactional embedded data manager for un-typed data in basic key/value data structures. This data manager stores all the data using pairs key-data. In oder to work with this API we have consulted the official tutorial provided by Oracle - Getting Started with Berkeley DB. [21]

3.4 Virtualization Technologies

The IaaS cloud-computing model has become feasible to be implemented when a free virtualization solution such as Xen Hypervirsor became available. Before virtualization, turning over the control to the Cloud resources to each client was endangering the system very much as users could easily subvert one of the Cloud's sites. The contribution virtualization brings into the Cloud is isolating the leased resources from the site in a secure way decreasing the risks brought by offering to users full control over their resources. Actually with virtualization giving control to users over their remote resources isn't any more a threat for the site owners. [7]

Other important assets that virtualization brings for the Cloud system is better mobility and flexibility. Also virtualization made possible dividing a machine's resources according to the client's demand.

3.4.1 Xen hypervisor

The Xen architecture is the most well thought-out virtualization technique on the market and can operate on a wide variety of architectures as well as older CPUS, without supporting full virtualization. Besides these advantages we have chosen to use Xen as a virtualization solution for the VMMs because is well integrated with the Unix systems and lately has made important steps to becoming a stable solution for cloud-computing, including Nimbus.

Xen boots from a bootloader like GNU GRUB and then usually loads a modified host operating system into the host domain (dom0). Xen under Linux runs on x86. Also Debian Lenny stable release version (the operating system we have used on all the Cloud's nodes) include in it's repository Xen 3.2.1. On Xen the machines are very easy to deploy, administer, access and modify. Each machine uses a configuration file when it is deployed which can contain a lot options – including networking configuration or how much RAM is allocated to a certain machine. Also Xen provides a network bridge which if used with a DHCP server can dynamically assign IP addresses to the virtual machines from a certain pool. This is an important feature on the cloud considering that when deploying the virtual machines into the Cloud is much expensive to use static IP addresses than randomly assigned ones.

3.4.2 Libvirt

Libvirt is an open-source management tool, for managing virtualized platforms like Xen or KVM. In the version 2.4 Nimbus uses the libvirt to manage virtual machines for both KVM or Xen virtualization solution. Libvirt uses the XML format for the VMM configuration files, this files being generated by the service provider when parsing the configuration files. [30]

3.5 Checkpoint-Restart Approaches

In order to be able to use BlobSeer into the Cloud we must be able to stop and restart it from a consistent state. In order to achieve this we have chosen to start implementing a checkpoint-restart module for BlobSeer. Checkpoint-restart is a method to backup certain states of the data, at certain moments – in order to be able to restart a software or bring a certain database to a consistent state. [10]

Many ordinary batch processes like backup and restore operations on servers are time-consuming. They consist of many units of work. If checkpointing is enabled, checkpoints are initiated at specified intervals, in terms of units of work or of processing time, depending on each application. At each checkpoint, intermediate results and a log recording the process's progress are saved to non-volatile storage. The contents of the program's memory area may also be saved.

The purpose of checkpointing is to minimize the amount of time and effort wasted when a long software process is interrupted by a hardware failure, a software failure, or resource unavailability. With checkpointing, the process can be restarted from the latest checkpoint rather than from the beginning. [22]

Providing failure-tolerance for shared-memory in distributed systems using checkpoints is a good method for long-running applications. The main achievement when using the checkpointing techniques in such environments is conserving the system's consistency in case of failure. There are three main approaches when implementing checkpointing in such systems: uncoordinated checkpoints, coordinated checkpoints, communication-induced checkpoints. [9], [10]

- Uncoordinated Checkpoints [10]

In the uncoordinated checkpoints method each entity determines its local checkpoint independently and at restart it searches the set of saved checkpoints for a consistent state to which it can resume. Besides some obvious disadvantages that this approach has (like the risk of the domino effect) this method wins by simplicity and good overall performance. Its main advantage is the autonomy of each entity to save its state in the most convenient moment. *Eg:* when certain time consuming operations are finished; when the state information to be saved is small;

- Coordinated Checkpoints [10]

In the coordinated checkpointing approach, processes must ensure that global checkpoints formed by local saved stated are guaranteed to be consistent. This is usually achieved by some kind of two-phase commit algorithm. The coordinated approach eliminates the risk of the domino effect [9] – as each process always restarts from its most recent consistent state. Apart from this advantages, this method's weakness is the big latency involved in saving the checkpoints – as a global checkpoint needs to be determined before checkpoints can be written to stable storage.

- Communication-Induced Checkpoints

This method which is also called dependency induced assumes that each process checkpoints its own state independently whenever this state is exposed to other processes (these are called local checkpoints) In order to achieve a global consistent state processes might need to take additional checkpoints (these are called forced checkpoints). The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring high latency and overhead. In contrast with coordinated checkpointing, no special coordination messages are exchanged in this approach. [10]

Checkpoints should occur frequently enough to minimize wasted effort when a restart is necessary but not so frequently as to prolong the process unduly with checkpoint overhead. Optimal checkpoint frequency depends on the mean time between failures (MTBF), among other factors.

Considering the assets and drawbacks of each approach for the BlobSeer project we have decided to implement the uncoordinated checkpoints method. This approach is the fastest way of all, as it can be done in convenient moments for each process doesn't have the overhead of assuring a global state.

4 Our contribution

4.1 BlobSeer on the Cloud – Setup Description

The global picture of the setup described in this chapter can be seen in the Fig 8. In these figure can be seen the main components of the Nimbus architecture, the means of communication between these components and commands can be addressed to each components. Furthermore in this architecture there are specified the main characteristic of all the Nimbus elements and what assets each must have. Besides Nimbus this figure also includes how BlobSeer will be deployed on the Cloud.

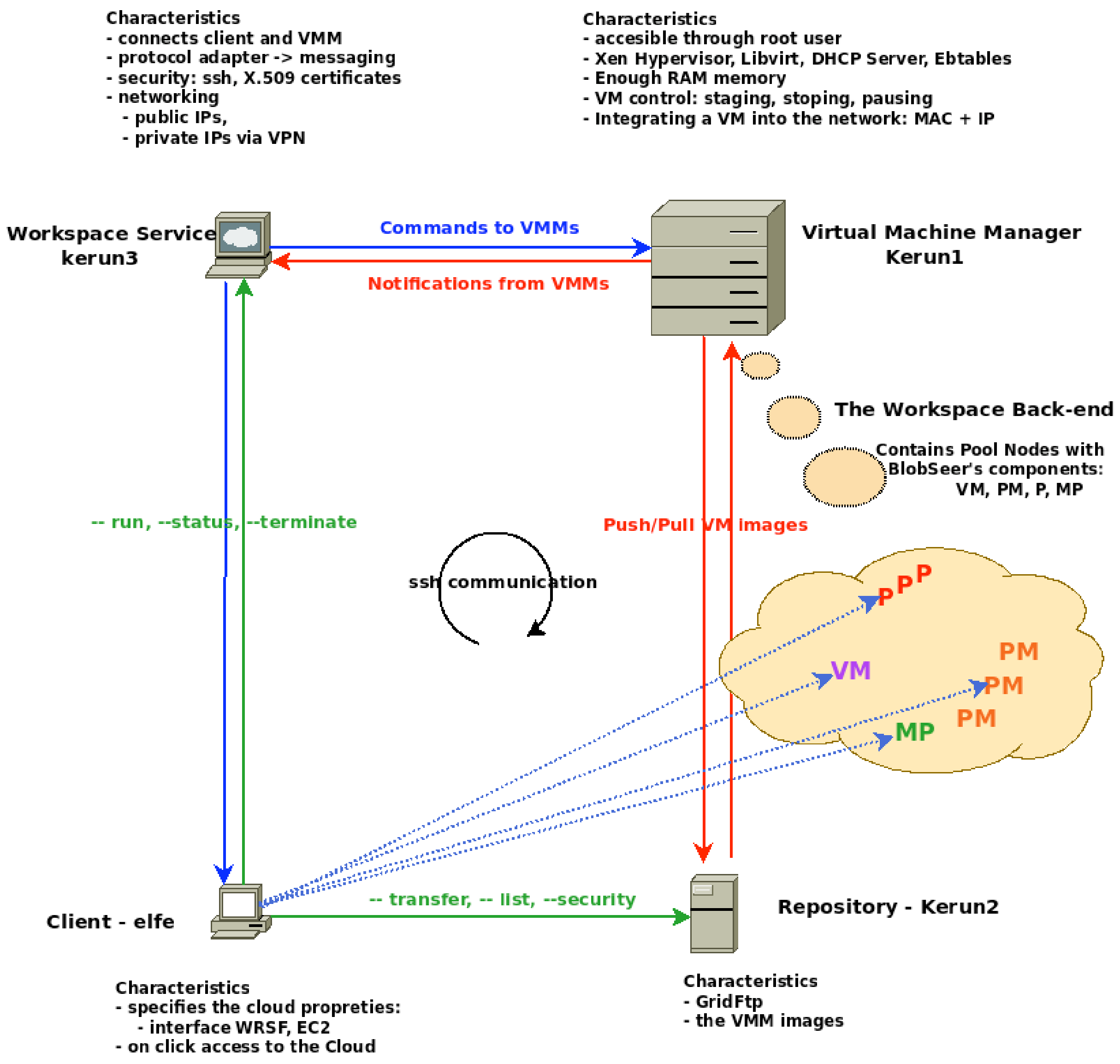
Characteristics
- connects client and VMM
- protocol adapter -> messaging
- security: ssh, X.509 certificates
- networking
- public IPs,
- private IPs via VPN
Characteristics
- accesible through root user
- Xen Hypervisor, Libvirt, DHCP Server, Ebtables
- Enough RAM memory
- VM control: staging, stoping, pausing
- Integrating a VM into the network: MAC + IP
Workspace Service kerun3
Commands to VMMs
Notifications from VMMs
Virtual Machine Manager Kerun1
The Workspace Back-end
Contains Pool Nodes with BlobSeer's components: VM, PM, P, MP
-- run, --status, --terminate
Push/Pull VM images
ssh communication
P
P
P
VM
PM
PM
PM
MP
Client - elfe
-- transfer, -- list, --security
Repository - Kerun2
Characteristics
- specifies the cloud propreties:
- interface WRSF, EC2
- on click access to the Cloud
Characteristics
- GridFtp
- the VMM images

Fig 8 Global Architecture – Nimbus

4.1.1 Nimbus Deployment

These description of the setup we have done will cover all the settings we had to modify or had some difficulties finding them. We started from the Cloud administration guide – Quickstart, version 2.4, on the Nimbus official site. We will give details for each of the Nimbus components.

4.1.1.1 The Client

The Cloud Client's purpose is to offer on click access to the Cloud for the end user from any kind of devices. Here we installed the the package Cloud Client 015 from the Nimbus official downloads web page.

The cloud client uses this credential to make contact with the service and the GridFTP server (which can be on separate nodes), sending the relevant commands to the relevant endpoint (authenticating anew with each new request) like in the Fig 8 - *Global Architecture – Nimbus*:

The X.509 authentication certificates for the clients can be found in the folder /home/student/.globus/ as follows:

- *hostcert.pem* - The host certificate. The certificate for the issuing CA must be in the Nimbus trusted-certs directory, in hashed format.
- *hostkey.pem* - The private key. Must be unencrypted and readable by the Nimbus user.
- *usercert.pem* – The certificate for the Nimbus user. In our deployment we have used nonroot user named student.
- *userkey.pem* – the private key for the user.

After installing this package and configuring the authorization certificates is important to know where the configuration and logging files can be found and what errors one might encounter on the road.

The configuration files for the client are located in $GLOBUS_LOCATION/conf folder.

The history (logs, configuration) files are located in $GLOBUS_LOCATION/history/vm-number. The only configuration file we have modified for the client is cloud.properties. The list with all the available options can be found on Nimbus client configuration page [28].

Next we will specify and explain the configuration option we have used in our configuration.

- `ssh.pubkey=~/.ssh/id_rsa.pub # the components of the cloud connect to each other using ssh and rsa public/private key pairs`
- `vws.factory=kerun3.hpc.pub.ro:8443 # the dns:port where the workspace service launch its services`

- `vws.repository=kerun2.hpc.pub.ro:2811 # the dns:port of the gridftp server`
- `vws.factory.identity=/O=Grid/OU=GlobusTest/OU=simpleCA-kerun2.hpc.pub.ro/CN=host/kerun3.hpc.pub.ro # the CA identity`
- `vws.cahash=cd97fdb1 # the hash for CA certificate`
- `vws.memory.request=300  # how much RAM memory is allocated to each virtual machine`
- `vws.metadata.mountAs=xvda1 # the divices for virtual machine are mounted differently than from a host machine (/dev/sda)`
- `vws.metadata.association=public # the name for virtual machines networking pool`
- `vws.metadata.vmmVersion=3 # the xen version`

We had an issue regarding the xvda1 option. Nimbus defaults to */dev/sda* based names, but this can be changed to the more recent *tap:aio* based *xvda* conventions as follows.

```
vws.metadata.mountAs=xvda1
```

In order to be able to communicate with the Repository and the Workspace Service the Client has to initialize the authentication based on X.509 certificates.

```
$NIMBUS_LOCATION/bin/grid-proxy-init.sh hours 240
```

With the following sets of commands we will be able to find out valuable information about the virtual machines deployed on the cloud.

- Commands to the Repository

The next command gives information about the used X.509 certificate:

```
student$ GLOBUS_LOCATION/bin/cloud-client.sh –security

/opt/nimbus-cloud-client-014/lib/certs

Credential in use:

  - Identity: '/O=Grid/OU=GlobusTest/OU=simpleCA-
kerun2.hpc.pub.ro/OU=hpc.pub.ro/CN=student'

  - Subject: 'O=Grid,OU=GlobusTest,OU=simpleCA-
kerun2.hpc.pub.ro,OU=hpc.pub.ro,CN=student,CN=1258301832'

  - Issuer: 'O=Grid,OU=GlobusTest,OU=simpleCA-kerun2.hpc.pub.ro,OU=hpc.pub.ro,CN=student'
```

The next command lists all the available virtual machines from the repository which are ready to be launched. This machines are built to be able to run on Xen.

```
$GLOBUS_LOCATION/bin/cloud-client.sh –list

/opt/nimbus-cloud-client-014/lib/certs

[Image] 'BlobSeer'                          Read/write

        Modified: Apr 23, 2010 @ 14:42   Size: 3250585600 bytes (~3100 MB)
```

- *Commands to the Workspace-Service*

In order to launch a certain machine the following commands can be used. This command will offer a lot of info regarding the machine's hardware and startup configuration, like networking.

```
$GLOBUS_LOCATION/bin/cloud-client.sh --run --name part2 -hours 240
```

4.1.1.2 The Workspace-Service

The main purposes of this component is to be a communication interface between the Client and the VMM, a messaging adapter. Here we have installed the nimbus service, version 2.4 from the official Nimbus project download page.

In the default install, Nimbus uses two files to manage authorization of users on the remote interfaces: users with X.509 credentials are checked against a grid-mapfile and users of the EC2 Query frontend are checked against the query users.txt list. In this setup we used X.509 credentials. In order to setup an access control list Workspace Service is using the *$GLOBUS_LOCATION/etc/nimbus/nimbus-grid-mapfile*. This file specifies which remote identities can access the container or a specific service. However one needs to ensure that any identity specified in this file is from a "trusted" Certificate Authority. CAs are trusted by placing their certificate in the trusted-certs directory (by default: "$NIMBUS_HOME/var/ca/trusted-certs/"). [29]

The configuration files we have modified for the Workspace Service component are located in the $CONF_GLOBUS folder.

```
export $CONF_GLOBUS=$GLOBUS_LOCATION/etc/nimbus/workspace-service.
```

All the settings are loaded when the globus-start-container is launched. If you change any settings you have to reload this server. Attention each time you reload this server it won't see the machines that were already started on the Workspace control. (the notifications from the Workspace Control won't get to this new instance of the Workspace Service).

In the $CONF_GLOBUS location there are files containing important settings for the workspace service:

- *logging.conf* Here we have set all the logging options to yes. $CONF_GLOBUS/logging.conf and the logs for each machine are located in $GLOBUS_LOCATION/var/nimbus. Here you can find information about accounting, reservations, caches, received notifications from the Workspace Control and Client
- *ssh.conf* In this file we set up root user to connect the Workspace Control: “control.ssh.user=root”
- *accounting.conf* The minimum amount of time the charge is rounded to 1 minute. charge.granularity=1
- *vmm-pools*: pool1

```
# node_name  memory_to_manage networks_supported
```

```
  kerun1     1524             *
```

The first column represents the Workspace Control name. If you have more workspace controls you can specify each in a row. The second field represents the maximum memory that can be occupied on the workspace control by the virtual machines. If third field is blank or * it is assumed that all networks are supported. When all the memory specified in the second column has already been used and the client wants to launch another machine on that Workspace Control it will receive the following error message.

```
$GLOBUS_LOCATION/bin/cloud-client.sh --run --name 12 -hours 240

Problem: Resource request denied: Error creating workspace(s): No resource pool has an applicable entry

Problem running 'vm-195'.
```

- *network-pools/public;*

we have configured the public pool configuration file using the private network 192.168.122.0/24 with the following elements in the pool:

```
pub02 192.168.122.12 192.168.122.1 192.168.122.255 255.255.255.0

pub03 192.168.122.13 192.168.122.1 192.168.122.255 255.255.255.0

pub03 192.168.122.14 192.168.122.1 192.168.122.255 255.255.255.0

pub03 192.168.122.15 192.168.122.1 192.168.122.255 255.255.255.0

pub03 192.168.122.16 192.168.122.1 192.168.122.255 255.255.255.0
```

– Deployment

After configuring everything on this component we should initialize the authentication based on X.509 certificates with.

```
$GLOBUS_LOCATION/bin/grid-proxy-init (-)hours 240
```

The next step is to start all the necessary services that would allow the client to communicate with the VMM.

```
$GLOBUS_LOCATION/bin/globus-start-container -debug
```

4.1.1.3 The Repository

This component's function is to store the VM images. Here we have installed GridFTP (an extension of the FTP standard) server from the Globus Website.

For starting the authentication process you have to run the next command.

```
$GLOBUS_LOCATION/bin/grid-proxy-init
```

The repository also uses X.509 certificates located at /home/student/.globus/simpleCA.

After finishing the configuration we only have to start the GridFTP server using

```
$GLOBUS_LOCATION/sbin/globus-gridftp-server &
```

4.1.1.4 The VMMs

The VMM node is the node with which manages all the virtual machines. The privileged account on the service node needs to be able to SSH freely to the VMM (without needing a password). And vice versa, the privileged account on the VMM nodes needs to be able to freely SSH back to the Workspace Service nodes to deliver notifications. On this node we have installed the Workspace-Control and the Xen Hypervisor.

The configuration files from the VMM are located in $GLOBUS_LOCATION/etc/workspace-control. This settings are loaded by the Workspace Service when a virtual machine starts.

The logs of workspace environment for each machine goes to $GLOBUS_LOCATION/var/workspace-control/logs. There are different logs per machine for each operation create/propagate/remove. The logs settings can be modified from logging.conf.

The virtual machines image are staged to $GLOBUS_LOCATION/var/workspace-control/secureimages.

We will specify here the most important settings we have used:

- *main.conf*

```
[vmcreation]

num_cpu_per_vm: 2
```

- *kernel.conf*

Instead of the default setting "authz_kernels: fake-vmlinuz-2.6-xen" the option "authz_kernels: vmlinuz-'uname -r'" should be use. Also if commented the setting "matchramdisk: -initrd" should be uncommeted.

- *libvirt.conf*

```
[libvirt]

vmm: xen3 # this specifies to the Workspace Service we are using Xen as a Virtualization
solution.

[libvirt_connections]

xen3: xen:///  # this is the string to which the virsh should connect
```

- *networks.conf*

Here are mentioned information about the bridge to create for each virtual machine. we have used the following settings.

```
[bridges]

eth0: 192.168.122.0/24

[dhcp-bridges]

# BRIDGENAME-CHOSEN: DHCPD-LISTENING-INTERFACE

eth0: eth0 #this means that the Domain 0 of the xen (de default domain) is listenting on
eth0  (in our case eth0:0) and this is also the interface on which the dhcp server is
listening.
```

As these settings are very important for being able to access the deployed virtual machines anwe will mention more details for the configuration.

On the Workspace constrol we have the following network settings:

```
student@kerun1 $/sbin/ifconfig

eth0 - inet addr:192.168.2.100  Bcast:192.168.2.255  Mask:255.255.255.0

eth0:0 - inet addr:192.168.122.1  Bcast:192.168.122.255  Mask:255.255.255.0

wrksp-158-0 // and a bridge for each of the powered on virtual machine

student@kerun1

$brctl show

bridge name     bridge id               STP enabled     interfaces

eth0            8000.001a921598c1       no              peth0

                                                        wrksp-158-0
```

For each started virtual machine an ebtables rule is added in order to only allow traffic to and from those virtual machines. Use ebtables -Ln to see what rules were added for your virtual machines.

- *xen.conf*

Depending on how is your virtual machine image created the option tap:io or file should be used for

```
[xencreation]

disk_driver:file
```

When all the configuration are done properly we will be able to:

a) Start the Xen network-bridge

```
sudo /etc/xen/scripts/network-bridge start
```

b) Start the Dhcp Server which give IP-s to the Virtual-Machines

```
sudo /etc/init.d/dhcp3-server restart
```

After deploying some virtual machines on the cloud the virsh tool can be used. This tool is provided by libvirt in order to list information about the virtual machines and to connect to these virtual mashines.

a) List the current started virtual-machines

```
student$ virsh list
```

b) Connect to a certain machine

```
student$ virsh console machine_name
```

It it important to mention that installing this Cloud involved a lot of research as the administration guide from the Nimbus official site only doesn't offer support for all the components. A helpful resource for this is the Nimbus administration discussion list.[32]

4.1.2 Virtualization Solution – Xen Setup

Because when deploying Nimbus we encountered some problems with Xen we will provide here what environments we have tested before getting to a stable working setup.

Because we wanted to have the latest stable version of the free Linux OS – we initially wanted to use Ubuntu 10.4 – but unfortunately this version doesn't include anymore xen-tools into its repository and configuring a kernel image especially for Xen requires having special hardware support for Xen. So installing Xen from it sources on Ubuntu 10.04 proved to be a bad choice.

Among other OS we have tested there is Ubuntu 8.04 which has almost all the needed packages in the repository. However there were some problems with this distribution also. Fortunately all these problems were resolved. An issue which we found difficult to solve was with the libvirt version, which was too old for the Nimbus setup.

Eventually we installed Xen on Debian latest stable version Lenny. Here we installed the following packages:

```
student$ dpkg -l | grep xen

libc6-xen, libxen-dev, libxenstore3.0, linux-headers-2.6-xen-686, linux-headers-2.6.26-2-
common-xen, linux-headers-2.6.26-2-xen-686, linux-modules-2.6-xen-686, xen-hypervisor-
3.2-1-i386,  xen-shell, xen-tools, xen-utils-3.2-1, xen-utils-common, xenstore-utils
```

After installing all the needed packages we have made the following configurations on the main configuration file xend-config.sxp, from /etc/xen.

```
(logfile /var/log/xen/xend.log)

(loglevel DEBUG)

(xen-api-server ((unix)))

(xend-http-server yes)

(xend-unix-server yes)

(xend-unix-path /var/lib/xend/xend-socket)

(network-script network-bridge)

(vif-script vif-bridge)
```

```
(dom0-min-mem 196)
```

```
(dom0-cpus 0)
```

When the configuration process was over the xend needs to be started. We also added this action to be done when the servers starts.

```
/etc/init.d/xend start
```

Libvirt is another tool that we need to install for the version 2.4 of Nimbus on the workspace control – the VMM. On Debian Lenny the libvirt version 0.4.6, which we downloaded from the repository had a bug. The file src/python/workspacecontrol/defaults/lvrt/lvrt_common.py had more then one function named createXML - and the Workspace-Control would get confused. The solution we found was renaming these functions. Because we were still receiving errors from the Workspace Service we dug deeper and found that the function that should have been named createXML was actually called createLinux. After repairing this we didn't had any more problems with libvirt.

4.1.3 BlobSeer Setup

For installing and deploying BlobSeer we followed the tutorials from the BlobSeer website. However as we encountered some problems along the way a more detailed tutorial which includes also a way to track problems during the installation and deploying process can be found at Annexe 1 – Installing and Deploying BlobSeer.

Each of these machines will have BlobSeer installed on them and will be configured to start with a certain instance on it. As a virtualization solution we have used Xen. In order to start each of the instances described in section 3.1.1 we need to do the followings:

Version Manager

```
$BLOBSEER_HOME/vmanager/vmanager localhost-config-file.cfg
```

Provider Manager

```
$BLOBSEER_HOME/pmanager/pmanager localhost-co4nfig-file.cfg
```

Provider

```
$BLOBSEER_HOME/provider/provider localhost-config-file.cfg
```

Metadata Provider

```
$BLOBSEER_HOME/provider/sdht localhost-config-file.cfg
```

After we have started all the BlobSeer components we are able to create BLOBs and to write/read to/from them with the following commands:

Create Blob

```
./create_blob /tmp/blobseer.cfg 8 1
```

Write to last created BLOB

```
./test W 1 /tmp/blobseer.cfg
```

Read from last created BLOB

```
./test R 1 /tmp/blobseer.cfg
```

Once the BlobSeer was on the cloud we needed an interface to be able to easily start and stop BlobSeer on the Cloud. We have managed to deploy this interface using bash scripting as presented in the chapter 4.3 Command Line interface for controlling BlobSeer in the Cloud. These instances communicate using ssh access thus between the BlobSeer machines we assure ssh access with public/private keys for the user student.

4.2 BlobSeer checkpoint-restart inside Nimbus Cloud

4.2.1 Existing approach on checkpoint restart in BlobSeer

BlobSeer is a strong application which besides having an efficient algorithm when working with huge amount of data also has the base for being able to restore the state of some elements after restarting them. The Provider Manager is a component that is able to restore its data after being restarted. This happens because the Provider is designed to send messages to the Provider Manager at certain intervals of time and this way the Provider Manager is able to regenerate the Providers table very fast after being restarted. The other components, Version Manager, Provider, Metadata Provider, of BlobSeer were not able to recover from a consistent state after restart, therefore we will describe next how we implemented support for incremental checkpoints on them.

4.2.2 Our approach on checkpoint restart in BlobSeer

BlobSeer is a storage application specially developed for storing data in distributed systems. This system provides support for working with huge amount of data divided into BLOBs identified through an unique Id, offers fine-grained access to the data, uses versioning, has very high throughput even under heavy access concurrency in any combination: read/read, read/write, write/write. However as this software is still in research state it has some shortcomings that needs to be solved.

BlobSeer was built as a very efficient storage application which works with data saved into the RAM memory (a volatile memory). A weak point that BlobSeer has into the Cloud is not being able to start from a consistent state. This application always starts working from zero data, so all the processed data are lost if the administrator decided not to use the Cloud for a period of time or if a system failure. In other words one lack that needs to be fulfilled before porting BlobSeer to the Cloud is implementing the support to be able the make scheduled restarts without losing the already processed data. One of the Clouds big advantage is that one can shutdown his application and stop using the Cloud when it doesn't need it without losing his current data. This is rather an important issue as the Cloud client pays for each rented byte and on the Cloud the user is able to get the Cloud's infrastructure on-demand. The client doesn't need to pay if in certain periods doesn't use the infrastructure. In this contexts being able to restart the application from a consistent state is a very important feature that BlobSeer needs to have when integrated with the Cloud. As a result in order to prepare BlobSeer for the Cloud we need to assure consistent states of this storage layer at certain moments in time. This will also help to make this file storage system more reliable to failure.

In order to assure starting BlobSeer from a consistent state we have considered a checkpoint-restart approach. Basically we will be storing a snapshot of the current application state – and later on we will use it for restarting the application from that consistent state. This restart can be done in case of failure or simply for administration decisions. Before choosing how to take these checkpoints we have considered the following aspects

- The BlobSeer's algorithm is based on achieving high performance when working with very big amounts of data. The key point of this algorithm is being able to do most of its operations in parallel using locking as little as possible. The method that would bring fault tolerance to BlobSeer shouldn't affect the great advantage BlobSeer brings, this being it's great speed when working with very big sets of data.
- Doing a snapshot adds the overhead of saving the data to a non-volatile memory. Performing to many snapshots could affect the system performances. This is why is better to be able to have a strict procedure for taking these snapshots. Therefore a good choice would be performing checkpoints when scheduled by an administrator of the system or automatically at fixed moments in time. Making to many snapshots of the system might contribute to making BlobSeer a slower application. Deciding when it's the best time to make a global checkpoint is quite difficult.
- The processes of BlobSeer run in parallel and need to communicate between them only when they've finished a read/write operation so we might admit that “most of the time” these entities run independently.
- As we have presented earlier in the *3.5 Checkpoint-Restart Approaches* chapter, there are various ways of doing snapshots – each having its advantages and disadvantages.

After acknowledging the points above we have decided that the best way to go first is choosing to implement “uncoordinated checkpoints”. This method implies that each process in the system will store each consistent state. Main advantages of this method are providing a good performance and in the same time the autonomy of each entity to “chose” the best moment to save its state. As a result this implementation also maintains the good performance the BlobSeer has without our module and also made it an application well integrated with the Cloud.

4.2.3 Solution implementation

When running BlobSeer into the Cloud system we need to develop new features like failure-tolerance the ability to start this application from a consistent state. In order assure starting BlobSeer from a consistent state before shutdown we have chosen to implement a incremental checkpoints for the reasons described in the previous paragraph. Next we will describe in detail how we have implemented the uncoordinated checkpoints on BlobSeer.

Next we will describe what modification we have done on each of the BlobSeer components:

- Provider and Metadata Provider

These two entities have to be added the functionality to save the data on the disk into a database at certain time intervals and when started first to load last saved data back into RAM. In order to do this we used the Berkley DB library and API as this provides high-performance embedded database with bindings also in C++, the language in which BlobSeer is developed.

BDB stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key. BDB can support thousands of simultaneous threads of control or concurrent processes manipulating databases as large as 256 terabytes, on a wide variety of operating systems including most Unix-like and Windows systems, and real-time operating systems. [21]

The functionality of restoring the keys from Berkley DB database is added in the template used in a failure-tolerant scenario *<bdb_bw_map>*. In this scenario BlobSeer uses a database to store the Provider and Metadata Provider data, data that is normally kept only into RAM. We used the support to save the data stored into RAM to the Berkeley DB database at certain time intervals and we added loading the data from that database back into RAM. In order to implements this store/load system we used the Berkley DB API because this type of database is very scalable and has good performances when working with big sets of data. To conclude for the Provider and the Metadata Provider we modifies the template *<bdb_bw_map>* that is used by both entities as a template when each of these processes is started. The code for this implementation is presented in the *9.2 Annexe 2 – Checkpoints for Provider and Metadata Provider.*

- Version Manager

Implementing the system described above on the Version Manager implies saving a serialized version of the data that currently is only kept in the RAM. Thus when the command to save the state is sent these parts save their serialized data into files. At restart the data is restored from those files back into RAM memory. As a consequence this entity is able to start working from the last created BLOB. The class that needs to be modified in order make the Version Manager able to restore its state before restart is the vmanagement class. In this class we have modified the method create, method responsible for creating the BLOB, and the constructor as described in the Anexe 2 – 9.2 Checkpoints for Version Manager.

- Provider Manager

As described in chapter *4.2.1 Existing approach on checkpoint restart in BlobSeer* – the provider already has a good way of regenerating the table with all the Provider Managers in the system. Consequently in this project we have just tested this functionality on the Cloud.

4.3 Command Line interface for controlling BlobSeer in the Cloud

In this section we will describe the interface we use to deploy the Nimbus Cloud "on click" from the end user side and how to run the BlobSeer instances with a single script after the Cloud was started.

In order to start the Cloud system one needs to run the script *./start_cloud.sh* from the Cloud client with the following arguments:

```
student@elfe:~$ ./cloud_deploy

-u student -r kerun2 -s kerun3 -c kerun1 -e elfe

All operations succesfully finished!


Usage: cloud_deploy options  -u <username> -r <repo_hostname> -s
<worskspace_service_hostname> -c <workspace_control_hostname> -e <end_user_hostname>
```

The complete content of this bash script can be seen in the *Annexe 3 – Scripts for Command Line interface for BlobSeer*. Behind this script there are the following operations:

- Begin authentication process based on X.509 certificates for the components Client, Workspace-Control, Workspace-Service.
- Start the GridFTP server which will make possible the transfer of the virtual machines from the client to the Repository and from the Repository to the VMM using the Workspace-Service.
- Run all the necessary services on the Workspace-Service in order to receive and run the commands received from the Client, receive update messages from the VMM and transfer them to the Client.
- Deploy the machines we have configured on the VMM. The deployed machines have 300 MB memory RAM, are from the network 192.168.122.0/24 and are in the range 192.168.122.12 – 192.168.122.20.

The operation of starting the cloud with 5 virtual machines usually takes from 15 to 20 minutes. This operation includes copying each machine from the Repository node to the VMM. On a real cloud this operation will surely last less as the networking protocol inside the Cloud would usually support better broadband.

As soon as the Cloud is started we can start the BlobSeer service on the Cloud by using the script $BLOBSEER_HOME/scrips/bl_cloud_deploy.sh. By running this script we are performing the following operations:

- Start the Provider Manager and Version Manager on the first virtual machine deployed on the Cloud.
- Start 2 Providers and 2 Metadata Providers on the virtual machine deployed second and third, respectively fourth and fifth.

- The Provider and Metadata Provider loads in their RAM the last written.
- The Version Manager loads information about all the BLOBs created previously restart.
- The Provider Manager will "discover" all the Providers in the system and this way will be able to reconstruct its provider table.

The action of deploying all the clouds components into the cloud depends on the ssh speed connectivity, which is very fast on the LAN (our case). The discovering protocol for the Provider Manager depends on the number of Providers in the system, as each Provider will send an acknowledge message each 5 seconds. The action of restoring for the Provider the last written data usually takes 30 seconds at restart.

If we need to shut down all the BlobSeer components we need to do run the script $BLOBSEER_HOME/scrips/bl_cloud_kill.sh. This script just does pkill on all the BlobSeer components that were deployed on the IaaS Nimbus Cloud.

We are also able to stop the virtual machines using the Cloud Client and to save the machine state with the same name, or with a new one (creating this way another xen image).

If we only need to start/stop one component from BlobSeer we can use the script available for each component in the $BLOBSEER_HOME/scripts folder from the client PC, <component>_kill/start.sh

After we received the "All done!" message the client is able to create BLOBs and to do write/read operations on them. In order to perform these operations the executables ./create_blob and /test W/R can be used exactly as they would be used on localhost. There is no need to concern about these operations as they use the configuration file /tmp/blobseer.cfg generated when all the components were started and discovered each other.

4.4 Tests

4.4.1 First Scenario: Deploying the first machine on the Cloud

For this scenario we have deployed the setup described in the chapter *4.1 Blobseer on the Cloud – Setup description* and tested the Cloud system by running a virtual machine with the networking and memory settings we have configured. Also an important action that needed to be tested was the ability to connect from the Client (*elfe*) on that virtual machine. As one of our goals was using private IPs (using public IPs for the virtual machines deployed on the Cloud is much more expensive than using random public IPs) we had to configure the client to be able to connect to the VM network through VPN.

The global picture of this test can be easily understood from the Fig 9

After deploying the virtual machine we are able to see the IP assigned to the machine by running:

```
student@elfe:~$ $GLOBUS_LOCATION/bin/cloud-client.sh --status

[*] - Workspace #181. 192.168.122.12 [ pub02 ]
```

```
      State: Running
      Duration: 14400 minutes.
      Start time: Fri Jul 02 05:31:14 EDT 2010
      Shutdown time: Mon Jul 12 05:31:14 EDT 2010
      Termination time: Mon Jul 12 05:41:14 EDT 2010
      *Handle: vm-213
       Image: 202
```

In order to connect to the machine we have just deployed we can just use ssh (as we are connected through VPN to that private network). Another important point is that if we run dhclient on that virtual machine it will receive the same IP – because the Workspace-Control grants a certain IP by matching it with the MAC address.

Step-by-step - Starting a machine into the Cloud

1. The command run by the Workspace Service when starting a machine on the Cloud. This command represents that the Workspace Service copies the solicited machine from the repository to the workspace control. (--propagate) with the following command:

```
/usr/bin/env python /opt/nimbus/src/python/workspacecontrol/main/wc_cmdline.py -c
/opt/nimbus/etc/workspace-control/main.conf --propagate --name wrksp-158 --images
scp://kerun2.hpc.pub.ro/cloud/5456a325/12 --notify
kerun3.hpc.pub.ro:22//home/student/instal_nimbus/services/var/nimbus/msg-
sinks/notifications
```

2. After propagation command the workspace instance is created

```
WORKSPACE INSTANCE CREATED:

    - Name: 'https://kerun3.hpc.pub.ro:8443/vm-192'

    - Start time:                Jun 30, 2010 7:14:59 AM

    - Shutdown time:             Jul 10, 2010 7:14:59 AM

    - Resource termination time: Jul 10, 2010 7:24:59 AM

    - Creator: /O=Grid/OU=GlobusTest/OU=simpleCA-
kerun2.hpc.pub.ro/OU=hpc.pub.ro/CN=student

    - ID: 160, VMM: kerun1
```

3. The machine is launched with the options gathered from the Client, Workspace Service and Workspace Control

```
/usr/bin/env python /opt/nimbus/src/python/workspacecontrol/main/wc_cmdline.py -c
/opt/nimbus/etc/workspace-control/main.conf --create --name wrksp-160 --memory 300
--networking  eth0; public; A2:AA:BB:9C:BB:48; Bridged; Static; 192.168.122.12;
192.168.122.1; 192.168.122.255; 255.255.255.0;192.168.2.1; pub04;null;null;null;null
--images file://13 --imagemounts xvda1 --mnttasks 7ad66523-bbb2-4475-
a3fd;/root/.ssh/authorized_keys
```

Step-by-Step - Save a machine with a different name is done by creating another machine image on the repository

1. The workspace service gracefully shutdowns your machine

2. Copies (--unpropagate) the modified machine with the new name if this is the case from the VMM to the Repository.

```
ssh -n -T -o BatchMode=yes root@kerun1 /opt/nimbus/bin/workspace-control.sh --unpropagate
--name wrksp-156 --images scp://kerun2.hpc.pub.ro/cloud/5456a325/part2 --unproptargets
scp://kerun2.hpc.pub.ro/cloud/5456a325/14 --notify
kerun3.hpc.pub.ro:22//home/student/instal_nimbus/services/var/nimbus/msg-
sinks/notifications
```

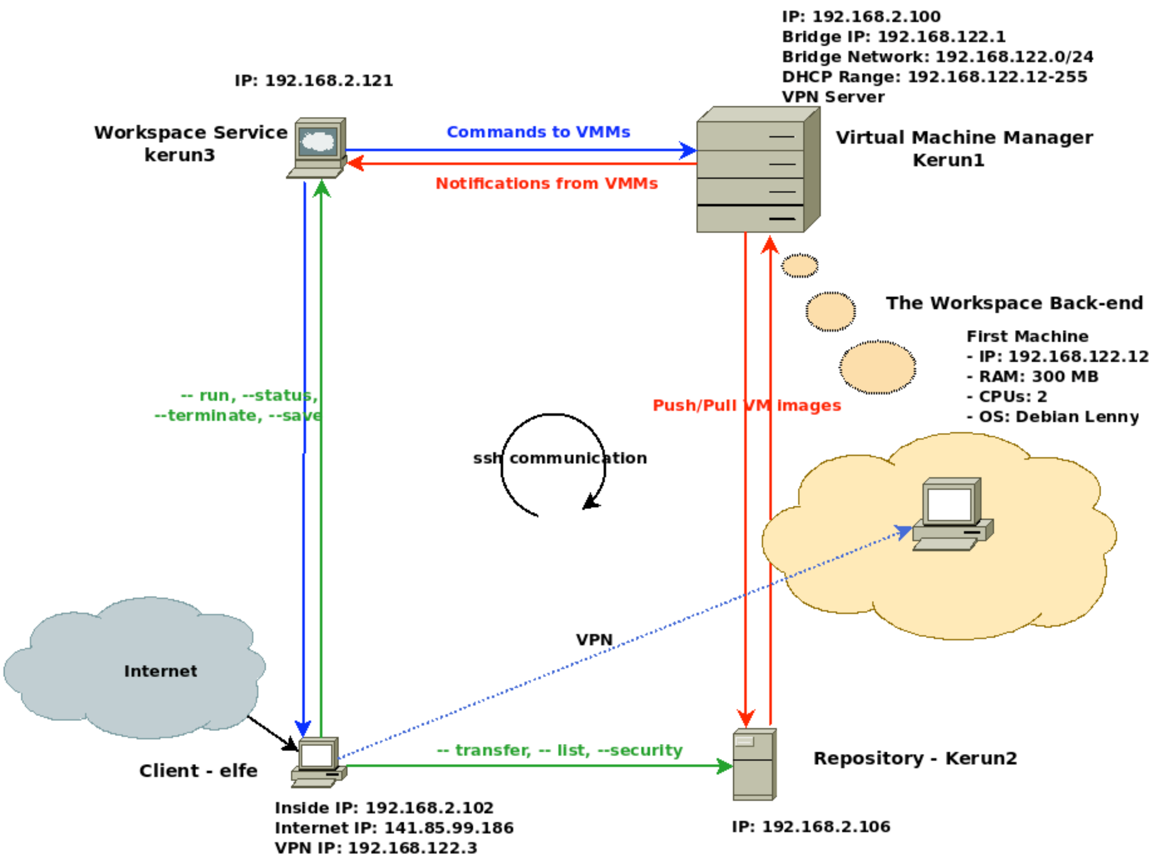

Fig 9 – Deploying the first virtual machine on the Cloud

4.4.2 Second Scenario: Starting more components of BlobSeer on the Cloud

This test is graphically represented in Fig 10. For this test we have started five virtual-machines on the Cloud and run the script ./cloud_deploy from the cloud client. The effect of this script was deploying a Version Manager an Provider Manager on the machine 192.168.122.12, two Providers on 192.168.122.13, 192.168.122.14 and two Metadata Providers on 192.168.122.15, 192.168.122.16.

```
./cloud_deploy.sh

192.168.122.12:

notroot@debian:~/blobseer/downloads/release-0.3.3/test$ netstat -tlpn

tcp        0 0.0.0.0:2222            0.0.0.0:*               LISTEN      4020/vmanager
tcp        0 0.0.0.0:1111            0.0.0.0:*               LISTEN      3895/pmanager

192.168.122.13-192.168.122.14:

notroot@debian:~/blobseer/downloads/release-0.3.3/scripts$ netstat -tlpn

tcp      0 0.0.0.0:1235            0.0.0.0:*               LISTEN      8421/provider

192.168.122.15-192.168.122.16:

notroot@debian:~/blobseer/downloads/release-0.3.3/scripts$ netstat -tlpn

tcp      0 0.0.0.0:1234            0.0.0.0:*               LISTEN      8276/sdht
```

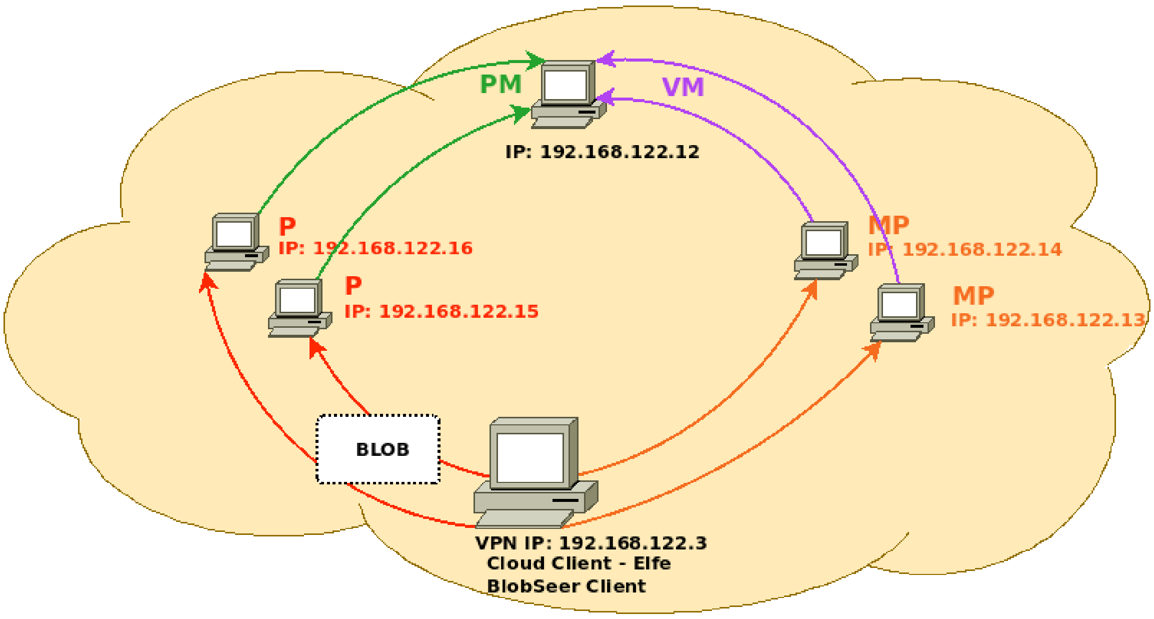

Fig 10 – Starting more components of BlobSeer on the Cloud

4.4.3 Third Scenario: Test Checkpointing for Provider/Metadata Provider

This test scenario is graphically represented in the Fig 11. For performing this test we have used the 5 virtual machines from the previous test. For testing that the Provider of Metadata Provider restores its data from the database, we have used a file to write the data the Provider/Metadata Provider has after restart, but before performing any read/write action thus proving that this entity loaded the keys from the database. This file is named “show_restored_data ” and we will use it only for testing.

```
192.168.122.13: ./provider_kill.sh

192.168.122.13: cat show_restored_data

empty file

192.168.122.13: ./provider_deploy.sh

192.168.122.13: cat show_restored_data

in while keie: (Size = '61', Data = '16 0 0 0 73 65 72 69 61 6c 69 7a 61 74 69 6f 6e 3a
3a 61 72 63 68 69 76 65 7 4 4 4 8 1 0 0 0 0 0 1 0 0 0 8 ffffffac fffffff4 ffffff9c f 2 0
0 0 0 0 0 8 0 0 0 0 0 0 0 0')

data:(Size = '8', Data = '3a 3a 3a 3a 3a 3a 3a 3a')

[...]
```

```
The data written before the provider was killed has been restored.
```

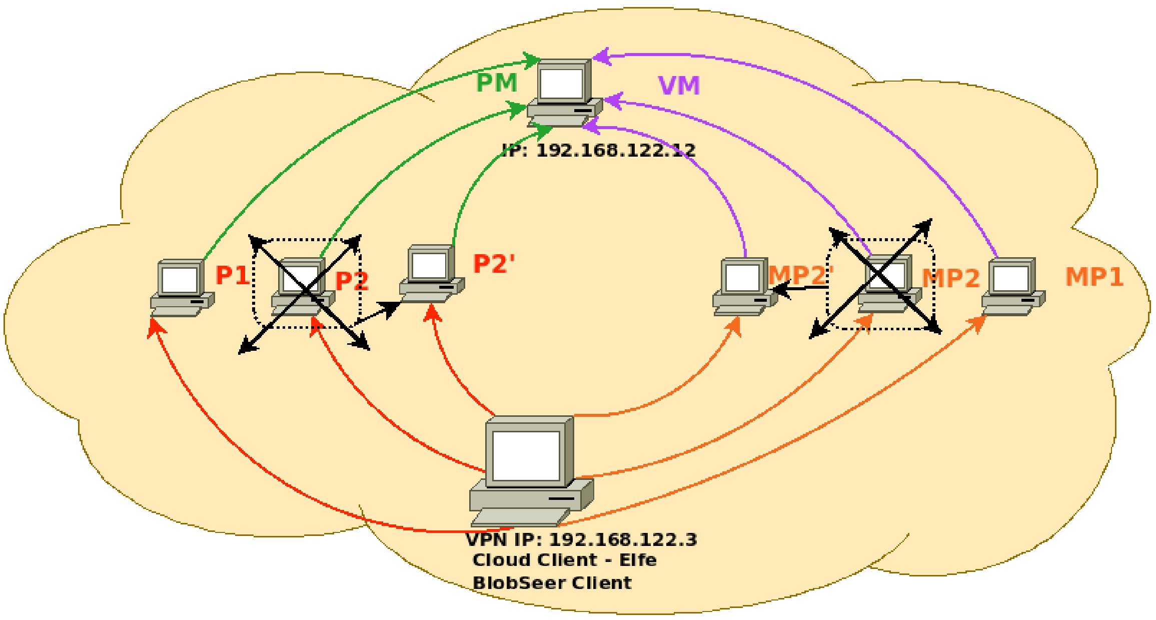

Fig 11 – Test Checkpoint functionality for Provider/Metadata Provider

4.4.4 Fourth Scenario: Test Checkpointing for Version Manager

This test is graphically represented in the Fig 12. In order to test the consistency of the Version Manager we have restated this component and checked if a new created blob was given an incremented from the value before restart.

Creating 2 BLOBs

```
notroot@debian:~/blobseer/downloads/release-0.3.3/scripts$ ./cloud-deploy.sh
notroot@debian:~/blobseer/downloads/release-0.3.3/test$ ./create_blob test.cfg 8 1
create result = 1, id = 1
Blob created successfully.
notroot@debian:~/blobseer/downloads/release-0.3.3/test$ ./create_blob test.cfg 8 1
create result = 1, id = 2
Blob created successfully.
```

Restarting Version Manager

```
notroot@debian:~/blobseer/downloads/release-0.3.3/scripts$ ./vm_kill.sh
kill version manager
remote_launcher ssh 192.168.122.12 "pkill vmanager || echo could not kill vmanager"
All done!
notroot@debian:~/blobseer/downloads/release-0.3.3/scripts$ ./vm_deploy.sh
```

```
x = 192.168.122.12

All done!
```

Creating third BLOB:

```
notroot@debian:~/blobseer/downloads/release-0.3.3/test$ ./create_blob test.cfg 8 1

create result = 1, id = 3

Blob created successfully.
```

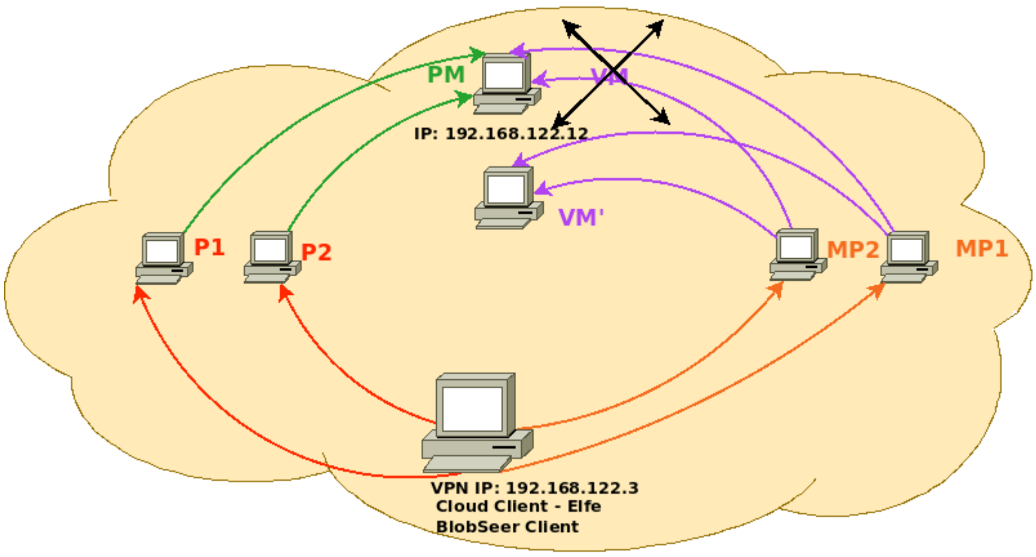

Fig 12 – Test Checkpointing for Version Manager

4.4.5 Fifth Scenario: Test Checkpointing for Provider Manager

This test scenario is graphically represented in the Fig 13. For performing this test we have started the followings providers 192.168.122.13 , 192.168.122.14 . We have done a write operation and the provider manager assigned data to be written by both providers. We have restarted the Provider Manager and it still continued to write to both providers, proving that the provider table was restored.

```
192.168.122.12: pkill pmanager

192.168.122.12: /home/notroot/blobseer/downloads/release-0.3.3/pmanager/pmanager
/tmp/blobseer.cfg >/tmp/blobseer/pmanager/pmanager.stdout
2>/tmp/blobseer/pmanager/pmanager.stderr&
```

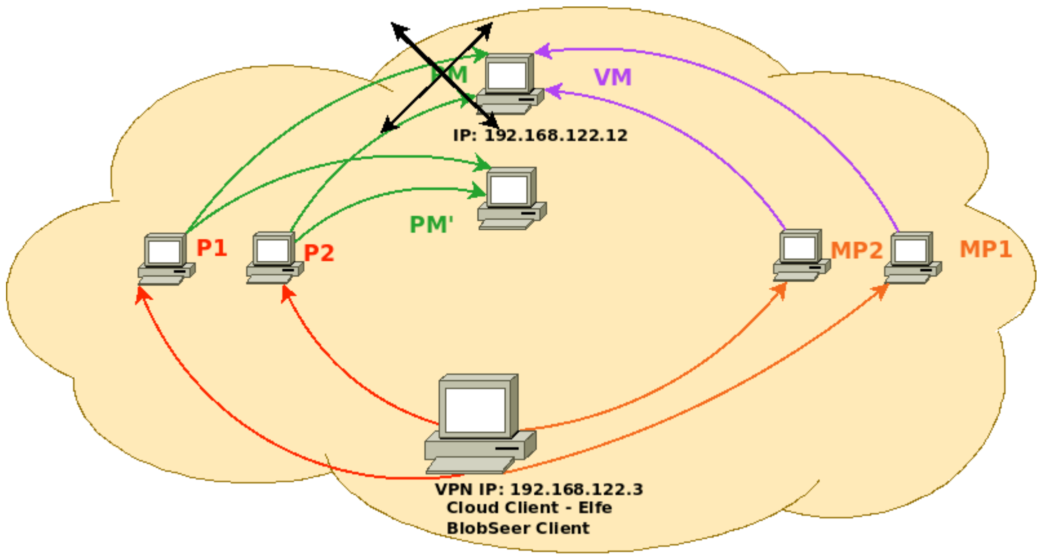

Fig 13 – Test Checkpointing for Provider Manager

5 Resources

The Cloud system is supposed to be installed on a large collection of servers from which there are used as many nodes as there are required by the Cloud user. However because the purpose of this project didn't include any load-balancing tests a laboratory environment was enough to test the integration between BlobSeer and Nimbus cloud system. Because we had encountered more challenges regarding the hardware and software resources we will describe here the resources we had available for this project.

5.1 Hardware

The hardware resources used to install the Nimbus components consisted of four computers with dual core CPU at 2.53 GHz and 2 GB of RAM. On each of these stations we have installed one component from Nimbus. These resources limited how many virtual machines we could deploy on the cloud and also how much RAM memory we could assign to each machine. Concerning the non-volatile memory we can say we had more than we actually needed. On the Repository, the component that used this memory to store the virtual machine images we had 300 GB and we only used 10 % of it.

Another important hardware related aspect concerns the networking resources. In this project we used a LAN with the bandwidth limited to 100 Mb/s. In a real Cloud the network performances could easily increase which will influence positively the time of deploying a virtual machine into the Cloud and the overall communication time between the Cloud elements.

5.2 Software

The operating system used on each Nimbus component including on the virtual machines with BlobSeer was the Debian Lenny 5.0 version, last release in January 2010. We have considered this version of Linux because it has proven stability and good community support. For installing Nimbus we have used the 2.4 version of this cloud toolkit which is very different from the 2.3 version and didn't have a well documented administration guide. As stated on the Nimbus administration discussion list the version 2.5 will have a lot of improvements which will make the set up process more intuitive.

For the Workspace Control component we have also tried Ubuntu 10.04 but we didn't manage to install Xen Hypervisor on it, because the xentools software was not longer maintained by an Ubuntu community developer, therefore it was removed from the Repository. We tried installing Xen from sources on this operating system – but we didn't succeed as we didn't have special hardware support for virtualization.

6 Conclusions and future developments

This project has opened a new development field for BlobSeer and brought it closer to become a viable alternative to its competitors as a storage solution for the Cloud system, GoogleFS, HDFS and Amazon S3.

BlobSeer is an application that serves as a storage layer when working with very big sets of data. As described in *3.3.2 BlobSeer's advantages on the Cloud* chapter its strong points that make a potential alternative for other similar application are the versioning system, achieving high throughput and the data-location-aware feature. The BlobSeer integration in the cloud-type system allowed us to test its behavior in this type of system and to review its weaknesses in the Cloud context. One of the BlobSeer's shortcomings was the fact that it would always loose its data on restart. Because the opportunity to stop and restart an application without losing the information processed in the Cloud system is very important, we decided to implement BlobSeer support to solve this lack. The approach on solving this issue was developing support for incremental checkpoint on BlobSeer for ths Provider, Metadata Provider and Version Manager. Because we had full control over the Cloud system we have managed to test and validate our solution in many test scenarios.

One of the important achievements of this project is also having a stable and complete setup of the Nimbus Cloud, which we can easily extend by adding more servers to play the role of VMM. On this environment we managed to deploy more virtual machines containing BlobSeer. We managed this by deploying and using some short scripts that allowed us the start and stop each of the BlobSeer components and therefore to observe that these components keep their data after restart. By implementing a system of incremental checkpoints for the BlobSeer components allows this application to come closer on becoming a stable software.

Cloud computing storage services is a topic of great interest in the context in which the Cloud computing model gains more and more followers. By making BlobSeer suitable for the Cloud model and making it a service for the Cloud we have proven that this application has great potential which besides achieving high performance is also an application with a high degree of adaptability.

The first step of porting BlobSeer on the Cloud lead to making BlobSeer more failure-tolerant and has proven that BlobSeer is a good match for this system. But there are still some important steps that need to be done before confirming this. One of this steps would be testing BlobSeer on a bigger cloud, which would allow us to perform some scalability tests. After performing these performance tests another important part of BlobSeer that has still place for development is the checkpointing system. Developing a model for making coordinated snapshots without affecting the BlobSeer's good performance would be another important step.

After we have tested the BlobSeer on the Cloud we thought to another more effective way to recover the Providers table of the Provider Manager by using broadcast messages when the Provider Manager restarts.

7 References

[1] Michael Miller, Cloud Computing Web-Based Applications That Change the Way You Work and Collaborate Online, Part I – Understanding Cloud Computing

[2] Paul Marshall, Kate Keahey and Tim Freeman “Elastic Site , Using Clouds to Elastically Extend Site Resources ”, The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 17 – 20 May 2010

[3] Judith M. Myerson IBM, Cloud computing versus grid computing - Service types, similarities and differences, and things to consider , 3 May 2009

[4] Bogdan Nicolae, Diana Moise, Gabriel Antonio – University of Rennes, Luc Bouje, Matthieu Dorier – ENS Cachan, Brittany , “BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications”, The 24th IEEE International Parallel and Distributed Processing Symposium, 19-23 April 2010

[5] Katarzyna Keahey, Tim Freeman “Contextualization: Providing One-Click Virtual” Clusters, 4th IEEE International Conference on e-Science, 11 December 2008

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung – Google, “The Google File System”, Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 20-43, 2003

[7] Katarzyna Keahey – University of Chicago, Mauricio Tsugawa, Andrea Matsunaga, Jose Fortes – University of Florida “Sky Computing “, IEEE Internet Computing Conference, pp 2-9, September 2009

[8] Cloud Slam 2010, Cloud Computing virtual conference http://cloudslam10.com, 23-25 May 2010

[9] E.N. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson, "A survey of rollback-recovery protocols in message-passing systems", ACM Computing Surveys , pp 2-7, September 2002

[10] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine - Open Systems Laboratory, Indiana University Parallel; Jason Duell, Paul Hargrove, Eric Roman - Lawrence Berkeley National Laboratory “Parallel Checkpoint/Restart for MPI Applications”, pp 1-4 , LACSI Symposium, October 2003

[11] Bogdan Nicolae, Advisor: Gabriel Antoniu, Renne France, Advisor: Luc Bouge BlobSeer ENS Cachan, Brittany: Efficient Data Management for Data-Intensive Applications Distributed at Large-Scale , 24th IEEE International Parallel and Distributed Processing Symposium, 19 – 23 April 2010

[12] HP CEO: The challenges of cloud computing,

http://www.zdnet.com/videos/events/hp-ceo-the-challenges-of-cloud-computing/355087, 21 October 2009

[13] http://aws.amazon.com/ec2/

[14] http://aws.amazon.com/s3/

[15] http://en.wikipedia.org/wiki/GlusterFS

[16] http://en.wikipedia.org/wiki/IBM_General_Parallel_File_System

[17] http://en.wikipedia.org/wiki/Parallel_Virtual_File_System

[18] http://www.globusconsortium.org/tutorial/ch6/page_5.php

[19] http://www.boost.org/

[20] http://www.hyperrealm.com/libconfig/

[21] http://www.oracle.com/technology/documentation/berkeley-db/db/gsg/CXX/index.html

[22] http://en.academic.ru/dic.nsf/enwiki/235617

[23] http://www.nimbusproject.org

[24] http://www.xen.org

[25] http://blobseer.gforge.inria.fr/doku.php

[26] http://hbase.apache.org/

[27] http://hadoop.apache.org/

[28] http://www.nimbusproject.org/docs/2.4/doc/cloud.html#configs

[29] http://www.nimbusproject.org/docs/2.4/admin/quickstart.html#part-IIb

[30] http://wiki.libvirt.org/page/Main_Page

[31] http://www.thoughtpolice.co.uk/vmware/

[32] workspace-user@globus.org

8 Annexe 1 – Installing and Deploying BlobSeer

As a notroot user in we installed all the requirements of the BlobSeer.

On the virtual-machine we have installed we used Debian Lenny (Debian 5.0), minimal (netinstall) version with 32-bits and 256 MB RAM, downloaded from a website with already installed vmware machines. [31]

Next we'll see some more detailed description.

```
root@debian:# cat /proc/meminfo

MemTotal:       256400 kB

root@debian:# cat /proc/cpuinfo

processor      : 0

vendor_id      : GenuineIntel

cpu family     : 6

model name     : Intel(R) Celeron(R) CPU          530  @ 1.73GHz

stepping       : 1

cpu MHz        : 1699.200

cache size     : 1024 KB

root@debian:# df -lh

Filesystem            Size  Used Avail Use% Mounted on

/dev/sda1             7.5G  1.7G  5.5G  24% /
```

Notes: There are two accounts: "root" and "notroot". Password for both is "thoughtpolice".

To complete the setup we have followed the following steps on the machine described above.

(a) Setting environment

```
notroot@debian:$ cd ~

notroot@debian:$ pwd

/home/notroot
```

We added in ~/.profile the following variables to be loaded on each login:

```
export HOME_BLOB=$HOME/blobseer/

export JAVA_HOME=/usr/lib/jvm/java-6-sun/

export HOME_BLOB=$HOME/blobseer/

export LD_LIBRARY_PATH=$HOME_BLOB/deploy/lib/

export BLOBSEER_HOME=$HOME_BLOB/downloads/release-0.3.3/

export CLASSPATH=$JAVA_HOME/bin/
```

We downloaded all the required tool for Blobseer in $HOME_BLOB/downloads/ and installed them in $HOME_BLOB/deploy/, so this tutorial can be started by creating this folders.

(b) **Boost -** a collection of free peer-reviewed portable C++ source libraries (Boost 0.37 and above). We used the version 1_42_0. You can go to http://sourceforge.net/projects/boost to check for the latest stable version and downloaded it. Afterwards we can run the following commands:

```
notroot@debian:$ cd $HOME_BLOB/downloads/

notroot@debian:$wget
http://sourceforge.net/projects/boost/files/boost/1.42.0/boost_1_42_0.tar.gz/download

notroot@debian:$ tar -xzvf boost_1_42_0.tar.gz

notroot@debian:$ cd $HOME_BLOB/downloads/boost_1_42_0/

notroot@debian:$        ./bootstrap.sh        --prefix=$HOME_BLOB/deploy        --with-
libraries=system,thread,serialization,filesystem,date_time -libdir=$HOME_BLOB/deploy/lib

notroot@debian:$ ./bjam

notroot@debian:$ ./bjam install
```

(c) **Libconfig -** a simple library for manipulating structured configuration files for C/C++. We downloaded the library from the following site:

```
notroot@debian:$ wget http://www.hyperrealm.com/libconfig/libconfig-1.4.4.tar.gz

notroot@debian:$ cd $HOME_BLOB/downloads/libconfig-1.4.3/

notroot@debian:$./configure --prefix=$HOME_BLOB/deploy && make && make install
```

(d) **Berkley DB -** Oracle Berkeley DB is the industry-leading open source, embeddable storage engine that provides developers a fast, reliable, local database with zero administration. To download it we can go to the next link and get the latest stable release http://www.oracle.com/technology/software/products/berkeley-db/index.html

We have downloaded and installed the version 4.8.26 with the following commands. This installation might take a while.

```
notroot@debian:$ cd $HOME_BLOB/download/sdb-4.8.26

notroot@debian:$ cd build_unix

notroot@debian:$ ../dist/configure  --prefix=$HOME_BLOB/deploy --enable-cxx

notroot@debian:$ make

notroot@debian:$ make install
```

(e) **Other necessary tools.** Beyond these requirements that are specified on the BlobSeer website to we had to install the followings:

Java6, using the steps from this toturial describing how to install Java6 on debian 5.0.

Subversion, from the repository with the command

```
notroot@debian:$ apt-get install subversion
```

CMake tool which will be used to build BlobSeer. We have installed it from the repository with

```
notroot@debian:$ apt-get install cmake
```

(f) **BlobSeer** – download the latest version directly from the repository with the command

```
svn checkout svn://scm.gforge.inria.fr/svn/blobseer/tags/release-**V**
```

To check the latest stable release go to Blobseer Svn .

BlobSeer is built using CMake – so this needs to be installed now if not done on the previous stage. Also before continue with compiling the **EXTERNAL_ROOT** environment variable needs to be set in the root prefix of dependencies section to this path. In the current setup example to set this variable we need to edit the CMakeLists.txt file (the tol-level configuration file) with the following commands:

```
notroot@debian:$ cd $HOME_BLOB/download/release-0.3.3/
```

edit CMakeLists.txt in order to set

```
EXTERNAL_ROOT= /home/notroot/blobseer/deploy/
```

Worth mentioning that are some preprocessor definitions used by BlobSeer. These definitions can be changed by editing the corresponding section in the configuration file. The user can control verbosity by defining **-D__INFO** for additional runtime information, as well as the socket type used for remote communications: **-DSOCK_TYPE=tcp** or **-DSOCK_TYPE=udp**. In this case we didn't edit any of these options. Afterwards we can proceed to the next step generating the make files, using a certain generator. In the current case we have used Unix makefiles. Once all Makefiles are build, we can proceed to build BlobSeer:

```
notroot@debian:$ cmake -G "Unix Makefiles"

notroot@debian:$ make
```

(g) If any errors happened during the built process a fully functional setup should be available. The next stage it's the deployment. BlobSeer is designed to be used in a large scale environment. A common deployment is performed in a local cluster, however, for a test environment the local machine is more than enough. This is why in the next example we will provide a local deployment example as this implies running all components of BlobSeer on localhost. In order to be able to create a blob and be able to write to it we need to start at very minimum *a version manager, a provider manager, a provider and a dht service provider*. Each of them reads a configuration file supplied as first parameter.

For this first test we can use from the BlobSeer sources downloaded from the SVN the file scripts/local-deploy.sh. You can run this script with the first parameter containing the configuration for each of the four entities. Normally there should be there an example of this file named blobseer-template.cfg. We have used the next configuration file in my setup:

```
notroot@debian:$cat  blobseer-template.cfg

# Version manager configuration

vmanager: {

    # The host name of the version manager

    host = ${vmanager};

    # The name of the service (tcp port number) to listen to

    service = "2222";

};

# Provider manager configuration

pmanager: {

    host = ${pmanager};

    service = "1111";

};

# Provider configuration

provider: {

    service = "1235";

    # Maximal number of pages to be cached

    cacheslots = 1024;

    # Update rate: when reaching this number of updates report to provider manager

    urate = 100;

    dbname = "/tmp/blobseer/provider/db/provider.db";

    #dbname = "";

    # Total space available to store pages, in MB (1GB here)

    space = 65536;

    # How often (in secs) to sync stored pages

    sync = 10;

};

# Built in DHT service configuration

sdht: {

    # Maximal number of hash values to be cached

    cacheslots = 100;

    # No persistency: just store in RAM

    dbname = "";
    # Total space available to store hash values, in MB (128MB here)

    space = 32;

    # How often (in secs) to sync stored hash values

    sync = 10;

};

# Client side DHT access interface configuration

dht: {

    # The service name of the DHT service (currently tcp port number the provider listens to)

    service = "1234";

    # List of machines running the builtin dht (sdht)

    gateways = (

        ${gateways}

    );

    # How many replicas to store for each metadata entry

    replication = 1;

    # How many seconds to wait for response

    timeout = 10;

    # How big the client's cache for dht entries is

    cachesize = 1048576;

};
```

This script uses blobseer-deploy.py in order to launch an entire environment with the minimum requirements specified above.

In order to make this script work fine we need to assure that the notroot user is able to login to localhost again without any password. For this we need to create private/public key pair and add the public key to ~/.ssh/authorized_keys.

You can use the next steps using ssh-keygen.

1. If not already installed openssh-server run:

```
notroot@debian:$ apt-get install openssh-server
```

2. Then create a pair of private/public keys with the command:

```
notroot@debian:$ ssh-keygen

Generating public/private rsa key pair.

Enter file in which to save the key (/home/notroot/.ssh/id_rsa):

/home/notroot/.ssh/id_rsa already exists.
```

```
Overwrite (y/n)? y

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/notroot/.ssh/id_rsa.

Your public key has been saved in /home/notroot/.ssh/id_rsa.pub.

The key fingerprint is:

ca:15:46:a9:57:5c:89:87:87:dc:e6:40:df:2e:06:19 notroot@debian
```

3. Then add the public key to authorized_keys with:

```
notroot@debian:$ cat ~/.ssh/id_rsa.pub > ~/.ssh/authorized_keys
```

4. If still having problems running the ./local_deploy.sh we should make sure that the variable $BLOBSEER_HOME is set up to the root where there is the downloaded BlobSeer from the SVN. To check this we can run one of the following commands:

```
notroot@debian:$ export //Shows all the exported variables

notroot@debian:$ echo $BLOBSEER_HOME // shows only the value of this variable in the
system.
```

If not set we can set it with the command (this command sets this variable only for the current login session to make this permanent add the command to ~/.profile which is read at every login.

```
export BLOBSEER_HOME=$HOME_BLOB/downloads/release-0.3.3
```

(h) After running `./local-deploy.sh blobseer-template.cfg` successfully (we should receive the message "All done!" In the console. To check that the version manager, the provider manager, the provider and the dht service provider were started we can run the command netstat and receive the following output.

```
notroot@debian:$ ~/blobseer/downloads/release-0.3.3/scripts$ netstat -tlpn

Proto Recv-Q Send-Q Local Address  Foreign Address State PID/Program name

tcp        0      0 0.0.0.0:2222   0.0.0.0:*       LISTEN      2071/vmanager

tcp        0      0 0.0.0.0:1234   0.0.0.0:*       LISTEN      2092/sdht

tcp        0      0 0.0.0.0:1235   0.0.0.0:*       LISTEN      2089/provider

tcp        0      0 0.0.0.0:1111   0.0.0.0:*       LISTEN      2079/pmanager
```

To stop BlobSeer we can run the script from the scripts folder `./local-kill.sh` and when we run the netstat command again the out above should not appear any more.

Back to deployment (if we stop BlobSeer we should start it again). When the start command is done there are some files created in /tmp like in the next output. There we can observe that each entity has it's own log file and the provider has a db with all the blobs and their version. Also directly into the /tmp folder it was generated the file blobseer.cfg which we will use in order to create blobs and read and write to them.

```
notroot@debian:$ ./local-deploy.sh blobseer-deploy.cfg

All done!

notroot@debian:$ cd /tmp

notroot@debian:$ tree .

├── blobseer
│   ├── pmanager
│   │   ├── pmanager.stderr
│   │   └── pmanager.stdout
│   ├── provider
│   │   ├── db
│   │   │   ├── __db.001
│   │   │   ├── __db.002
│   │   │   ├── __db.003
│   │   │   ├── __db.004
│   │   │   └── provider.db
│   │   ├── provider.stderr
│   │   └── provider.stdout
│   ├── sdht
│   │   ├── sdht.stderr
│   │   └── sdht.stdout
│   └── vmanager
│       ├── vmanager.stderr
│       └── vmanager.stdout
└── blobseer.cfg
```

(i) **Create first BLOB.** In order to create a BLOB we need to run the command create_blob from the test folder being `$BLOBSEER_HOME`.

```
notroot@debian:$ pwd

notroot@debian:$ /home/notroot/blobseer/downloads/release-0.3.3/test

notroot@debian:$ ./create_blob

Usage: create_blob <config_file> <page_size> <replica_count>. To be used in empty
deployment to generate ID 1.

notroot@debian:$ ./create_blob /tmp/blobseer.cfg 8 1

Blob created successfully.
```

```
End of test
```

(j) **Write/Read from a BLOB.** After creating a BLOB first thing we need to do is write to it and then we can append data to it and read from it. This can be done with the following commands

```
notroot@debian:$ pwd

notroot@debian:$ /home/notroot/blobseer/downloads/release-0.3.3/test

notroot@debian:$ ./test W 1 /tmp/blobseer.cfg

notroot@debian:$ ./test R 1 /tmp/blobseer.cfg
```

(k) **Problems encountered with standard deplyment for this setup**. By default in the deployment from BlobSeer website the page size goes to 256 RAM. My setup was on machines that had only 256 MB memory RAM. The solution to this was to resize the page dimension to 8KB. To do this we modified in test.cpp the following lines:

```
const uint64_t PAGE_SIZE = 1 << 3; // 8KB instead of 64 KB
const uint64_t START_SIZE = 1 << 3; // 8KB instead of 1 MB
const uint64_t STOP_SIZE = 1 << 16; // 16MB instead of 256 MB
```

and appender.cpp where we modified the following line:

```
onst boost::uint64_t PAGE_SIZE = 1 << 3;
```

we built the Blobseer again like in the step (f) and run the command from (i) and (j) successfully.

Another problem which we have encountered was running the script ./local-kill.py . The problem was caused by the fact that in debian the command killall is not istalled by default. We e replaced the “killall” with “pkill” in the file blobseer-deploy.py and everything went well with this command also.

9 Annexe 2 – Incremental checkpoints on BlobSeer.

9.1 Checkpoints for Provider and Metadata Provider

The Provider and Metadata Provider use the template <bdb_bw_map> when running the processes for these two components as seen in the next code sample from the main routine.

```
if (db_name != "")
      run_server<bdb_bw_map>();
   else
      run_server<null_bw_map>();
```

In this template the save operation is done in the method sync_handler at certain time intervals after the write operation was finished.

```
void bdb_bw_map::sync_handler() {
        boost::xtime xt;
        buffer_wrapper key, value;


        for (;;) {
                // now start the DB sync; make this operation uninterruptible
                boost::this_thread::disable_interruption di;
                while (1) {
                        {
                                scoped_lock lock(write_queue_lock);
                                if (write_queue.empty()) {
                                        break;
                                }
                                key = write_queue.front().first;
                                value = write_queue.front().second;
                                write_queue.pop_front();
                        }
                        Dbt db_key(key.get(), key.size());
                        Dbt db_value(value.get(), value.size());

                        try {
                                db->put(NULL, &db_key, &db_value, 0);
                        } catch (DbException &e) {
                                ERROR("failed to put page in the DB, error is: " <<
e.what());
                        }
                }
                try {
                        db->sync(0);
                } catch (DbException &e) {
                        ERROR("sync triggered, but failed: " << e.what());
                }}
```

We added the functionality of restoring the data from the database back into the RAM memory using the Berkley DB API for reading data from the database. After reading this data we serialize it and put it load it back into cache.

```
int bdb_bw_map::restore_records(Db &providerDB) {

        Dbc *cursorp;
        ofstream data_show;

        // temporary file, used only for testing reasons
        data_show.open("show_RESTORED_data", ios::app);

        try {
                providerDB.cursor(NULL, &cursorp, 0);

                Dbt key, data;
                buffer_wrapper *key_buf, *value_buf;
                int ret;

                // Iterate over the inventory database, from the first record
                // to the last, displaying each in turn
                while ((ret = cursorp->get(&key, &data, DB_NEXT)) == 0) {
                        // the buffers where each key is loaded
                        key_buf = new buffer_wrapper[1];
                        value_buf = new buffer_wrapper[1];

                        key_buf[0] = buffer_wrapper((char *) (key.get_data()),
key.get_size());
                        value_buf[0] = buffer_wrapper((char *)
(data.get_data()),data.get_size());
```

```
                      // saving the read data – only for testing reasons
                      data_show<<"cheie: "<<key_buf[0]<<endl<<"data:"<<value_buf[0]<<
endl;
                      buffer_wrapper_cache->write(key_buf[0], value_buf[0]);
              }
       } catch (DbException &e) {
              providerDB.err(e.get_errno(), "Error in restore_records");
              cursorp->close();
              throw e;
       } catch (std::exception &e) {
              ERROR("other error while accessing db");
              cursorp->close();
              throw e;
       }

       data_show.close();
       cursorp->close();
       return (0);
}
```

This restore method is called into the bdb_bw_map constructor before performing any operation with:

```
bdb_bw_map::bdb_bw_map(const std::string &db_name, boost::uint64_t cache_size,
boost::uint64_t m, unsigned int to) :
       […]
       bdb_bw_map::restore_records(*db);
       […]
```

9.2 Checkpoints for Version Manager

Storing incremental checkpoints on the Version Manager implies storing the following information for each created BLOB <id , page size, replica count >. Because the amount of information is small, only three integers per BLOB, we decided to use files to store this information.

The storing operation is done in the method that deals with creating each node from the class vmanagement:

```
rpcreturn_t vmanagement::create(const rpcvector_t &params, rpcvector_t &result,
              const std::string &sender) {
```

```
        ofstream obj_count_file;
        obj_count_file.open("obj_count.txt", ios::app);
        […]
        unsigned int id = ++obj_count;
        buffer_wrapper *my_obj;
        my_obj = new buffer_wrapper[1];

        //id, page_size, replica_count
        obj_info new_obj(id, ps, rc);
       // insert the pairs into the RAM memory
        std::pair<unsigned int, obj_info> to_insert_obj(id, new_obj);
        obj_hash.insert(to_insert_obj);
        result.push_back(buffer_wrapper(new_obj.roots.back(), true));
        // store the pair <id, page size, replica count> into a file on the disk
        obj_count_file << id << " " << ps << " " << rc << endl;}
        // close the file
        obj_count_file.close();
        return rpcstatus::ok;
}
```

The restoring operation is done in the Version Manager constructor by reading each pair data of the previously created BLOB and loading it into the RAM memory.

```
vmanagement::vmanagement() :
        read_count() {
        ifstream obj_count_file;
        int count = 0;
        unsigned int id;
        boost::uint64_t ps;
        boost::uint32_t rc;

        obj_count_file.open("obj_count.txt");
        // if this file exists we can restore the created BLOBs
        if (obj_count_file.good()) {
                        while (!obj_count_file.eof()) {
                                // reading the pair <id, page_size, replica_count>
                                obj_count_file >> id >> ps >> rc;
                                // loading this data into RAM
                                obj_info new_obj(id, ps, rc);
                                std::pair<unsigned int, obj_info> to_insert_obj(id,
new_obj);
                                obj_hash.insert(to_insert_obj);
                        // increment object number
                        obj_count++;
                }
        }
        // if not the default BLOB has the id equal to zero
        else {
                obj_count = 0;
        }
        }
```

10 Annexe 3 – Scripts for Command Line Interface for BlobSeer

10.1 Interface for deploying the Nimbus Cloud

In this section we will present the content of the script which starts our cloud.

start_cloud.sh:

```
#!/bin/bash

NO_ARGS=0

E_OPTERROR=85

if [ $# -eq "$NO_ARGS" ]    # Script invoked with no command-line args?

then

echo "Usage: `basename $0` options  -u <username> -r <repo_hostname> -s
<worskspace_service_hostname> -c <workspace_control_hostname> -e <end_user_hostname>"

exit $E_OPTERROR

fi

while getopts "u:r:s:c:e" Option

do case $Option in

u ) user=$OPTARG;;

r ) repo=$OPTARG;;

s ) service=$OPTARG;;

c ) control=$OPTARG;;

e ) enduser=$OPTARG;;

esac

done

echo "the arguments are user = $user, repo = $repo, service=$service, control=$control,
enduser=$enduser"

#start authentification process

ssh $user@$repo '$GLOBUS_LOCATION/bin/grid-proxy-init'

if [$? -ne 0]


   echo "error when starting the authentication process on repository node"

   exit 0

fi

#start GridFTP server

ssh $user@$repo '$GLOBUS_LOCATION/sbin/globus-gridftp-server &'

if [$? -ne 0]

   echo "error when starting GridFTP server on repository node"

   exit 0

fi
```

```
# on workspace-service -> kerun3
# start the authentication process
ssh $user@$service '$GLOBUS_LOCATION/bin/grid-proxy-init -hours 240'
if [$? -ne 0]
   echo "error when starting authentification process on workspace service node"
   exit 0
fi
# start all the workspace-service servicies
ssh $user@$service '$GLOBUS_LOCATION/bin/globus-start-container -debug'
if [$? -ne 0]
   echo "error when starting services on wrospace service"
   exit 0
fi
# on vmm -> kerun1
# start xen-network bridge
ssh $user@$control 'sudo /etc/xen/scripts/network-bridge start'
if [$? -ne 0]
   echo "error when starting authentification process on workspace service node"
   exit 0
fi
# restart DHCP server
ssh $user@$control 'sudo /etc/init.d/dhcp3-server restart'
if [$? -ne 0]
   echo "error when restarting dhcp server"
   exit 0
fi
# on client -> elfe
# initiate the authentification process
$GLOBUS_LOCATION/bin/cloud-client.sh --run --name part2 -hours 240
#start five virtual machines with BlobSeer on the IaaS Cloud
ssh $user@$enduser $GLOBUS_LOCATION/bin/cloud-client.sh --run --name 202 -hours 240 &
if [$? -ne 0]
   echo "error starting vm 202"
   exit 0
fi
ssh $user@$enduser $GLOBUS_LOCATION/bin/cloud-client.sh --run --name 203 -hours 240 &
if [$? -ne 0]
   echo "error starting vm 203"
   exit 0
fi
```

```
ssh $user@$enduser $GLOBUS_LOCATION/bin/cloud-client.sh --run --name 204 -hours 240 &
if [$? -ne 0]
   echo "error starting vm 204"
   exit 0
fi
ssh $user@$enduser $GLOBUS_LOCATION/bin/cloud-client.sh --run --name 205 -hours 240 &
if [$? -ne 0]
   echo "error starting vm 205"
   exit 0
fi
ssh $user@$enduser $GLOBUS_LOCATION/bin/cloud-client.sh --run --name 206 -hours 240 &
if [$? -ne 0]
   echo "error starting vm 206"
   exit 0
fi
echo "All done!"
```

10.2 Interface for deploying BlobSeer as a service on the IaaS Nimbus Cloud

In order to deploy and kill BlobSeer with all its instances we have used the following scripts. For starting/stopping single components of BlobSeer on the Cloud we have used similar scripts with the only difference that we have used blobseer-deploy.py script with different arguments.

./bl_cloud_deploy.sh

```
#!/bin/bash

VM=/tmp/deploy_blobseer/vm

PM=/tmp/deploy_blobseer/pm

P=/tmp/deploy_blobseer/p

MP=/tmp/deploy_blobseer/mp

TEMPLATE_FILE=$BLOBSEER_HOME/scripts/blobseer-template.cfg

echo "192.168.122.12" > $VM

echo "192.168.122.12" > $PM

echo "192.168.122.12" > $P

echo "192.168.122.13" >> $P

echo "192.168.122.14" >> $P

echo "192.168.122.15" > $MP

echo "192.168.122.16" >> $MP

./blobseer-deploy.py --vmgr=`cat $VM` --pmgr=`cat $PM`
--dht=$MP --providers=$P --launch=$TEMPLATE_FILE

rm -rf /tmp/deploy_blobseer/*
```

./bl_cloud_kill.sh

```
#!/bin/bash
```

```
VM=/tmp/deploy_blobseer/vm

PM=/tmp/deploy_blobseer/pm

P=/tmp/deploy_blobseer/p

MP=/tmp/deploy_blobseer/mp


echo "192.168.122.12" > $VM

echo "192.168.122.12" > $PM

echo "192.168.122.12" > $P

echo "192.168.122.13" >> $P

echo "192.168.122.14" >> $P

echo "192.168.122.15" > $MP

echo "192.168.122.16" >> $MP


$BLOBSEER_HOME/scripts/blobseer-deploy.py --vmgr=`cat $VM`
--pmgr=`cat $PM` --dht=$MP --providers=$P --kill


rm -rf /tmp/deploy_blobseer/*
```

./blobseer-deploy.py

```
#!/usr/bin/env python


REMOTE_CFG_FILE="/tmp/blobseer.cfg"


def usage():

    print "%s [-v, --vmgr <hostname>] [-m, --pmgr
<hostname>] [-d, --dht <dht.txt>] [-p, --providers
<providers.txt>] [ [-l, --launch] | [-k, --kill] | [-s,
--status] ]\n" % sys.argv[0]


def read_hosts(file_name):

    f = open(file_name, "r")

    hosts = map(lambda x: x.strip(), filter(lambda x: x !=
"\n", f.readlines()))

    f.close()

    return hosts


def get_env(name):

    if name in sys_vars:

        return sys_vars[name]

    try:

        sys_vars[name] = os.environ[name]

    except KeyError:

        print "WARNING: environment variable " + name + "
is not set"

        sys_vars[name] = "."

    return sys_vars[name]


def gen_launch(path, process):

    out = "/tmp/blobseer/" + process

    return "\"rm -rf " + out + "; " + \

        "mkdir -p " + out + "; " + \

        "env CLASSPATH=" + get_env("CLASSPATH") + " " + \

        "LD_LIBRARY_PATH=" + get_env("LD_LIBRARY_PATH") + "
" + \

        get_env("BLOBSEER_HOME") + "/" + path + "/" +
process + " " + \

        REMOTE_CFG_FILE + " >" + out + "/" + process +
".stdout 2>" + out + "/" + process + ".stderr&\""


def gen_kill(path, process):

    return "\"pkill " + process + " || echo could not kill
" + process + "\""


def gen_status(path, process):

    return "\"ps aux | grep " + path + "/" + process + " |
grep -v grep | wc -l | grep 1 >/dev/null || echo no " +
process + " running\""


class RemoteCopy(threading.Thread):

    def __init__(self, n, src, dest):

        threading.Thread.__init__(self)

        self.node = n

        self.cmd = SCP + " " + src + " " + self.node + ":"
+ dest + " &>/dev/null"

    def run(self):

        p = subprocess.Popen(self.cmd, shell=True)

        if p.wait() != 0:

            print "[%s]: could not copy configuration file"
% self.node


class RemoteLauncher(threading.Thread):

    def __init__(self, n, cmd):

        threading.Thread.__init__(self)

        self.node = n

        self.cmd = SSH + " " + self.node + " " + cmd
    def run(self):

        p = subprocess.Popen(self.cmd, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE)

        (cout, cerr) = p.communicate()

        if cout.strip() != "":

            print "[%s]: %s" % (self.node, cout.strip())


(opts, args) = getopt.getopt(sys.argv[1:], "v:m:p:d:l:ks",
["vmgr=", "pmgr=", "providers=", "dht=", "launch=", "kill",
"status"])

gencmd = gen_status

for (o, a) in opts:

    if o in ("-h", "--help"):

        usage()

        sys.exit(2)

    elif o in ("-v", "--vmgr"):

        vmgr = a

    elif o in ("-m", "--pmgr"):

        pmgr = a

  elif o in ("-p", "--providers"):

 providers = read_hosts(a)

    elif o in ("-d", "--dht"):

        dht = read_hosts(a)

    elif o in ("-l", "--launch"):

        gencmd = gen_launch

        f = open(a, "r")

        template = f.readlines()

        f.close()

    elif o in ("-k", "--kill"):

        gencmd = gen_kill

    elif o in ("-s", "--status"):

        pass

if vmgr == "" or pmgr == "" or providers == [] or dht ==
[]:

    usage()

    sys.exit(1)

# If needed, copy all configuration files to the remote
nodes

if (gencmd == gen_launch):

    gateways = ",\n\t\t".join(map(lambda x: "\"" + x +
"\"", dht))

    f = open(TMP_CFG_FILE, "w")

    f.writelines(map(lambda x: x.replace("${gateways}",
gateways).

                     replace("${vmanager}", "\"" + vmgr +
"\"").

                     replace("${pmanager}", "\"" + pmgr +
"\""), template))

    f.close()


    cfgs = set([vmgr]) | set([pmgr]) | set(providers) |
set(dht)

    for x in cfgs:

        workers.append(RemoteCopy(x, TMP_CFG_FILE,
REMOTE_CFG_FILE))


    for w in workers:

        w.start()


    for w in workers:

        w.join()

workers.append(RemoteLauncher(vmgr, gencmd("vmanager",
"vmanager")))

workers.append(RemoteLauncher(pmgr, gencmd("pmanager",
"pmanager")))

for p in providers:

    workers.append(RemoteLauncher(p, gencmd("provider",
"provider")))

for d in dht:

    workers.append(RemoteLauncher(d, gencmd("provider",
"sdht")))


for w in workers:

    w.start()


for w in workers:

    w.join()


print "All done!"
```