Nebu

A Topology-Aware Deployment System for Reliable Virtualized Multi-Cluster Environments

Jesse Donkervliet
Tim Hegeman
Stefan Hugtenburg

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

Nebu

A Topology-Aware Deployment System for Reliable Virtualized Multi-Cluster Environments

by

Jesse Donkervliet
Tim Hegeman
Stefan Hugtenburg

in partial fulfilment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Tuesday July 8, 2014 at 2:00 PM.

Supervisor:	Asst. Prof. dr. ir. A. Iosup	TU Delft
	V. van Beek MSc. ft.	Bitbrains
	Dr. ir. Martha Larson	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

bitbrains IT SERVICES

TU Delft Delft University of Technology

PAN EUROPEAN dca PROJECT
ESTABLISHING AN ACTION PLAN FOR THE DATA CENTRE INDUSTRY

Summary

Petabytes of data are processed daily by distributed applications built upon Hadoop and MongoDB. A significant fraction of these applications use cloud infrastructure to cope with this vast amount of data. Commercial clouds use virtualized environments, but most distributed applications are designed around the idea that they run on physical hardware. When this is no longer the case, guarantees for an application's reliability and performance no longer hold.

To remedy this issue, we design a powerful and comprehensive system called Nebu. Nebu is able to provide information about the physical topology of the cloud to the distributed application and is capable of automated virtual machine and application deployment. Nebu performs these tasks without depending on any single distributed application or virtual machine manager. Instead, Nebu provides efficient APIs that make it easy to provide compatibility with many popular distributed applications and virtual machine managers.

We develop Nebu as an open source project using modern software engineering practices. In particular, we use the agile development method Scrum in combination with the Kanban scheduling system. We apply iterative API design through the use of RAML and supporting UML diagrams. Because no effective methods for testing distributed applications have been developed, we use both unit testing and manual testing. We also apply automated regression testing through the use of continuous integration.

Because there are no formal guidelines on how to validate distributed applications for the kind we investigate in this work, we develop Nebu and perform real-world experiments with multiple distributed applications using an enterprise multi-cluster infrastructure. These experiments show that Nebu enables applications to give guarantees about reliability without degrading their performance.

To increase Nebu's usability, we provide extensions that offer compatibility with the distributed applications Hadoop and MongoDB, and virtual machine manager VMware. Both the system and the extensions to the system are developed in an enterprise environment. This holds good promise that Nebu will be adopted by open-source communities, as well as the industry.

Acknowledgements

This thesis would not have been concluded without the help provided by many people, a few of which we would like to recognise here. First of all Alexandru Iosup and Vincent van Beek for their excellent guidance and feedback on matters conceptual and technical. In addition we would like to thank them for going through multiple iterations of this lengthy thesis. Secondly, Gjalt van Rutten for allowing us to do this thesis at Bitbrains and helping us define the specifics of this project. Third, Otto Visser for his endless supply of coffee and suffering through the drafts of the documents we produced for this thesis. Finally we would like to thank everyone at Bitbrains and in the Parallel and Distributed Systems group for their warm welcome in their midst and the feedback after the trial presentations.

Contents

List of Figures

List of Tables

1

Introduction

1.1. Context

With an increasing number of companies providing cloud solutions to the industry and individual, the popularity of running applications in virtualized environments is on the rise. Similarly, distributed (big data) applications are getting increasingly popular. While these two trends can be combined to run distributed applications in a highly-scalable virtualized environment, there are several drawbacks for the current state-of-the-art. In particular, the reliability and fault tolerance of data storage by the system can no longer be guaranteed. Traditionally, fault tolerance mechanisms have been based on the assumption that the failure of a single node in a system is not correlated with the failure of others. In virtualized environments, this assumption no longer holds. Multiple virtual machines can be placed on a single physical machine and will all fail simultaneously if the physical machine fails.

Bitbrains is a Dutch company providing cloud solutions to a variety of customers, and they are facing the challenges of running distributed applications on their multi-cluster cloud infrastructure. Applications hosted by Bitbrains include Hadoop and MongoDB, both of which feature distributed data storage, with data replication as their primary means of fault tolerance. These applications spread several replicas of the same data over multiple virtual machines. As the applications are not currently aware of the virtualized environment, replicas of a chunk of data may end up on multiple virtual machines on the same physical host. If the physical host fails, then data may become unavailable or could even be lost. For enterprise customers serviced by Bitbrains, reliability is top priority and losing data can be catastrophic.

1.2. Problem Statement

The challenge set out by Bitbrains is to improve the reliability of distributed applications in a virtualized environment. To achieve this, Bitbrains is interested in developing a system that can utilise information about the physical topology of their multi-cluster environment to make distributed applications aware of the virtual environment, i.e., become "virtualization aware". We define the physical topology as the physical organisation of hosts to form racks and the grouping of racks into data centres. The goal is to develop a generic system that can interface with various virtual machine managers and distributed applications to prevent having to redevelop the solution every time customers demand a new application. As a proof of concept, the solution should be able to interface with VMware and OpenStack as virtual machine managers, and Hadoop and MongoDB as distributed applications.

The goals set by Bitbrains lead to the following research questions:

1. How to use physical topology information of a multi-cluster environment to improve the reliability of virtualized applications?

2. How to validate the design of the final product for distributed applications?

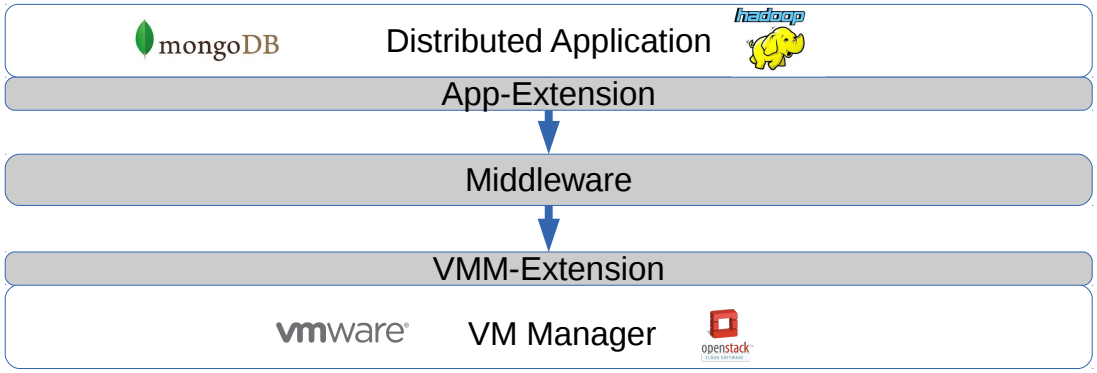


Figure 1.1: Simplified overview of the Nebu ecosystem, showing the three main components implemented by Nebu in grey.

1.3. Main Contributions

This thesis presents the Nebu project, an effort to improve the reliability of distributed applications running in a virtualized environment. Nebu is a software ecosystem consisting of three "classes" of applications. First of all, the Nebu core is a middleware application that operates between distributed applications and virtual machine managers, and interfaces with both to achieve the goal of improved reliability. Second, a virtual machine manager needs an extension to expose required information, such as virtual machine and topology information, to the Nebu core. Third, a distributed application needs an extension to use the information exposed by the Nebu core to improve its reliability.

The main contributions of this thesis are:

1. A user study through interviews that indicates what kind of functionality is required from the system by engineers at Bitbrains. These interviews form the basis for the use cases that lead to the API design presented in this thesis.

2. Nebu, a generic open source software ecosystem (visualised in Figure 1.1) to run distributed applications with virtualization awareness. This includes: (a) a generalised model for the physical topology of multi-cluster environments; (b) two APIs for extending the ecosystem with additional compatibility for virtual machine managers or distributed applications; and (c) an extensible application deployer with customisable placement policies.

3. An efficient API for engineers and end-users to create a virtual cluster and deploy a distributed application. Using Nebu's API it is possible to launch a complete cluster in four calls, or retrieve an overview of the physical topology in a single call. Currently the deployment of a complete cluster is a labour-intensive task that can take several hours to complete.

4. An evaluation of the impact Nebu has on both the reliability and performance of multiple big data platforms. Experimental evaluation is used to estimate the chance of failure in traditional settings, without virtualization awareness. Failure injection is used to identify the effects of failure in both traditional and virtualization aware scenarios.

1.4. Structure

This thesis is structured as follows: Chapter 2 gives a more detailed analysis of the problem as well as a description of the system requirements. Chapter 3 provides background information and presents related work. The processes used in this project, such as the usage of the Scrum methodology are described in Chapter 4. The design of Nebu is covered in Chapter 5 and the development of the final product is described in Chapter 6. Measures taken to increase the quality of the software, such as the writing of testcode and the usage of code analysis tools, are described in Chapter 7. In order to verify the final system, experimental work has been done. The results of these experiments can be found in Chapter 8 of this thesis. Finally, the future of Nebu is described in Chapter 9. Figure 1.2 provides a map guiding a reader through this thesis depending on the reader's goals.

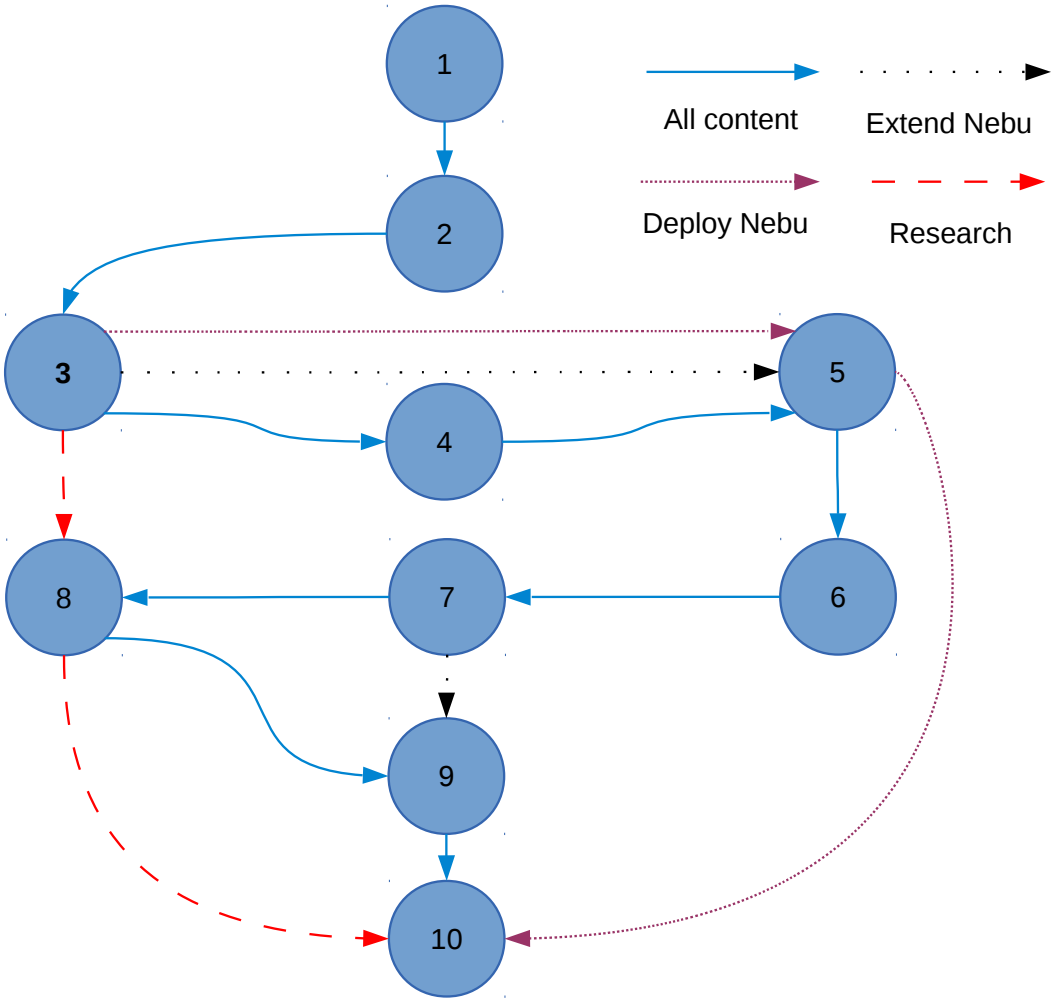


Figure 1.2: A visual representation of the chapters the reader should read if they have a specific goal in mind. If a chapter has no arrow of their colour exiting it, the "All content" arrow should be followed instead. For instance if the reader wants to deploy Nebu, chapters 1, 2, 3, 5, and 10 should be read.

2

Problem Analysis

2.1. Overview

With the rapid adoption of cloud computing, distributed applications are increasingly deployed in virtualized environments. However, most distributed applications were designed under the assumption that they are deployed on physical hosts, i.e., every machine running the application is a unique physical machine. This assumption is used in placement policies for various mechanisms, e.g., a fault tolerance mechanism places data replicas on different machines under the assumption that when one hard drive fails, other copies of the data are still available on other physical disks. In virtualized environments the assumption of uniqueness does not hold; multiple virtual machines may be running on the same hardware. As a result, multiple (or all) replicas of one piece of data may reside on the same physical hard drive. If this hard drive fails, the data may no longer be available, despite the application's fault tolerance mechanisms.

At Bitbrains, customers can lease machines from a virtualized environment, and many customers use these machines for distributed computing tasks. To prevent loss of data and to improve performance, Bitbrains would like to provide their customers with extensions for popular distributed applications that add VM-awareness. The goal of the project is to design an abstraction layer between VM managers and distributed applications to stimulate VM-awareness in distributed applications independent of the cloud it is running in. To demonstrate the applicability of this middle layer, extensions for MongoDB and Hadoop will be written on the application side, and a VMware extension will be written on the VM manager side. These application extensions improve the fault-tolerance and performance of a virtualized deployment of their respective distributed application when used in combination with a VM manager extension. The original problem description is included in Appendix C.

2.2. System Requirements

Throughout this project, system requirements have been identified before and during development. These requirements are either derived from the research questions listed in Section 1.2, or obtained through interviewing Bitbrains employees. Multiple requirements were changed or dropped during the project. In this section only the most up-to-date requirements are described. For initial requirements see the Orientation Report in Appendix A. For requirements that have been dropped or changed see Section 4.4.2. Table 2.1 provides an overview of the system requirements. The remainder of this section describes each Nebu requirement in detail.

2.2.1. Critical Requirements

Compatibility

Many distributed (big data) applications can be run in virtualized environments. Making these applications virtualization aware by providing them with physical topology information all works in a similar manner. Therefore, implementing this for every distributed application is undesirable. To prevent this, compatibility with applications should be defined in terms of an API that is offered to the application. An application is considered compatible with the system if it utilises this API.

#	Critical	Type	Requirement
1	YES	Compatibility	System must define a generic model of the physical topology in a multi-cluster environment.
2	YES	Compatibility	System must define an API that is extensible for other distributed applications.
3	YES	Compatibility	System must define an API that is extensible for other virtual machine managers.
4	YES	Deployment	System is capable of automated virtual machine deployment.
5	YES	Deployment	System is capable of configurable virtual machine placement.
6	YES	Topology awareness	System provides information about the location of virtual machines to distributed applications.
7	YES	Usability	System should be compatible with enterprise environments.
8	NO	Security	System features a authorisation system that ensures no other users can access data that is not their own.
9	NO	Usability	System should provide at least two application extensions as a proof of concept.
10	NO	Usability	System should provide at least two virtual machine manager extensions as a proof of concept.

Table 2.1: System requirements.

Multiple popular hypervisor or virtual machine manager (VMM) systems exists. These systems can differ in design, but conceptually provide the same functionality. Examples of these systems are VMware, OpenStack, and OpenNebula. Because multiple VMMs are commonly used, it is desirable that Nebu is compatible with multiple popular virtual machine managers. As with the distributed applications, this compatibility should be defined in terms of an API. This API should be offered to Nebu by the VMM. This API should be generic to ensure compatibility across hypervisors, but remain flexible enough to perform tasks such as virtual machine deployment and placement.

Deployment

Deploying virtual machines is a labour intensive task. When dealing with distributed applications, multiple virtual machines need to be deployed and configured for the application to work. To boost the productivity of a Nebu user, Nebu should automate this deployment based on a user-specified configuration. This allows the user to focus on more important tasks and leave the tedious task of deploying and configuring multiple similar machines to Nebu.

Virtualisation awareness provides applications with the tools to increase their performance and fault-tolerance. Improving these aspects consists of two major parts. First, the virtual machines should be placed on physical hosts by some placement policy. Second, the application improves its performance and fault-tolerance using the information about virtual machine placement. Depending on the type of application or the application's workload, different placement algorithms might be desirable for this task. Because Nebu is unaware of the application's properties and the properties of its workload, Nebu should leave this up to the user and provide a method through which the VM placement can be influenced.

Topology Awareness

Commercial clouds often lease virtual resources to their customers. Distributed applications that are unaware of the fact that they are running in a virtual environment can experience difficulties when providing guarantees for performance or fault-tolerance. Two machines that the application identifies as independent might actually be placed on the same physical host. When this host crashes, both machines go down. To provide distributed applications with the tools to prevent this, Nebu should expose partial physical topology information to the application. It is the responsibility of the application to make use of this information.

Usability

Running distributed applications on virtualized hardware is primarily done in commercial environments. To provide compatibility with these environments, Nebu should support enterprise environments and use external libraries that do the same.

2.2.2. Other Requirements

Security

Commercial clouds often have multiple tenants. All of these tenants are potential Nebu users and have access to the functionality Nebu offers. However, it is undesirable for these users to be able to see and modify deployments of other users. Nebu should provide an authorisation system where information is available on a per-user basis.

Usability

Because Nebu functions as a middleware layer between virtual machine managers and distributed applications, it relies on these systems to show its potential. To provide a proof-of-concept for Nebu's capabilities and increase Nebu's usability, extensions should be implemented for two distributed applications as well as extensions for two virtual machine managers.

3

Background

3.1. Overview

This section provides additional information needed to understand the details of the Nebu project. First, Section 3.2 provides additional information on some typical systems used in virtualized distributed computing. Second, Section 3.3 describes several state-of-the-art systems in resource management, and virtualization awareness.

3.2. Systems used by Nebu

The Nebu system provides extensions for both distributed applications and virtual machine managers. As a result, many external systems are involved in the project. To achieve requirements 9 and 10, two distributed applications and two virtual machine managers have been selected to receive proof-of-concept extensions. The two distributed applications selected for extension are MongoDB and Hadoop, introduced in Section 3.2.1 and Section 3.2.2, respectively. VMware and OpenStack are two popular virtual machine managers that have been considered for extension. They are introduced in Section 3.2.3 and Section 3.2.4, respectively.

3.2.1. MongoDB

MongoDB[1] is a distributed document database, using a NoSQL-structure for information retrieval and alteration. In contrast to the table-based model of SQL-databases, MongoDB utilises a dynamic JSON-like document store. MongoDB has several features to support its use as a distributed application, including replication and sharding. Replication is MongoDB's fault tolerance mechanism used to ensure high availability of data. By placing multiple copies of a single data set on multiple machines, data will remain available even if a machine crashes or becomes otherwise unusable. Additionally, replication can be used to spread data geographically and thus reduce latencies to applications deployed worldwide. Sharding is a technique used by MongoDB to achieve horizontal scaling, i.e. spreading data and load over multiple hosts. A single data set is split into shards and each shard is placed on a different machine. Queries on the data set are redirected to the relevant shards through a query router and results are then aggregated. Replication and sharding can be combined: a shard can be replicated to multiple machines to form its own replica set.

A typical distributed MongoDB database consists of three types of services: a set of shards (replica sets), three configuration (config) servers, and one or more query routers. The shards are spread over different data nodes, though multiple shards can co-exist on a single machine. The full collection of shards in the database is managed by three config servers. The config servers store the cluster's metadata, i.e., they keep track of the mapping of the cluster's data to the various shards in the database. For fault-tolerance, having three config servers is recommended, to deal with hardware failures. Finally, query routers are tasked with analysing queries and rerouting queries to the appropriate shards. Having many query routers is recommended, as all queries must pass through a query router. Having too few query routers can cause this step in execution to become a bottleneck.

[1] http://www.mongodb.org/

3.2.2. Hadoop

Hadoop[2] is an open source project by Apache that encompasses a stack of several distributed applications. First, it contains the Hadoop Distributed File System (HDFS) [1]. In contrast to the database structure that MongoDB uses, HDFS operates on files and employs a master/slave architecture that allows for file-system-like operations, e.g., opening, closing, and renaming files. Every file is split up in blocks and every block is replicated a configurable amount of times. Like MongoDB, HDFS uses replication of data to provide fault tolerance and high availability. The second component is YARN [2], a resource manager and job scheduler. YARN is responsible for managing resources in a cluster and can schedule distributed applications to use these resources. The third component of the Hadoop stack is its implementation of the MapReduce framework [3]. Hadoop MapReduce runs on top of YARN and HDFS to enable big data processing using the MapReduce programming model.

A typical Hadoop deployment consists of several services for HDFS and YARN. HDFS uses a NameNode as master and any number of DataNodes as slaves. YARN likewise uses a ResourceManager as master and any number of NodeManagers as slaves. Slave nodes commonly run both a DataNode and a NodeManager daemon, so that both the disk and compute resources of a node are utilised. Both HDFS and YARN provide mechanisms for improving their robustness. However, these mechanisms are not further explored in this thesis.

3.2.3. VMware

VMware[3] is a software company that provides software for cloud management and virtual machine management. VMware offers a large virtualization ecosystem with many layers of applications. At the core is the hypervisor, known as VMware ESXi, which is responsible for running virtual machines on physical hardware. Multiple ESXi hosts can be combined into a network that is managed by a vCenter instance. Each network managed by such a vCenter instance is known as a Virtual Data Center (VDC). VDCs can be combined to form the vCloud, an overarching system that enables end users to manage their virtual machines through a graphical user interface.

3.2.4. OpenStack

Similarly to VMware, OpenStack[4] also offers software that allows for virtualization management. This open-source variant deploys a modular architecture, with Nova, or OpenStack compute, at its core. Whereas Nova is responsible for the resource pools, other modules such as Swift and Neutron exist with other responsibilities (storage and networking respectively). The OpenStack API has also been made compatible with the well-known cloud provider Amazon's Elastic Compute Cloud (EC2)[5].

3.3. Related Work

Both distributed applications and virtualization have gained popularity in recent years. With companies moving to "the cloud", guaranteeing the reliability and performance of applications in virtualized environments have been a hot topic in research. In this section some related work will be described and compared to Nebu.

3.3.1. Hadoop Virtualization Extensions

To facilitate deploying Hadoop on top of VMware-based clouds, VMware has developed software known as VMware Big Data Extensions as described in a white paper[6]. The software provides Hadoop with information about the physical architecture underlying the virtual machines the application is running on. To achieve this, VMware has extended the popular Hadoop MapReduce framework with the Hadoop Virtualization Extensions to utilise the mapping of virtual to physical machines. This extension adds a single layer to the Hadoop network topology that describes the physical nodes VMs are deployed on. In addition, VMware has extended the block placement policy in HDFS to prevent it from placing multiple replicas of the same block on a single physical host.

[2] http://hadoop.apache.org/
[3] http://www.vmware.com/
[4] https://www.openstack.org/
[5] https://aws.amazon.com/ec2/
[6] http://www.vmware.com/files/pdf/Hadoop-Virtualization-Extensions-on-VMware-vSphere-5.pdf

VMware has made available an open source variant of their Big Data Extensions, known as Project Serengeti. Although Project Serengeti offers some similar functionality to Nebu in the virtualization awareness of Hadoop, the project is closely coupled to VMware and Hadoop. Nebu is designed from scratch to be extensible with any virtual machine manager or distributed application.

3.3.2. Mesos

Apache Mesos [4] seems to offer similar functionality at first glance, by presenting itself as a platform for sharing clusters between multiple distributed computing frameworks. Mesos requires computing resources to be assigned to it, so that it can deploy distributed applications on these resources. Unlike Nebu, Mesos will not interface with virtual machine managers to obtain additional resources. In addition, Mesos does not have explicit support for virtualization awareness. Although both Mesos and Nebu share the goal of running distributed applications in multi-cluster environments, Nebu is specifically targeted at virtualized environments.

3.3.3. YARN

YARN is another resource manager by Apache. As mentioned in Section 3.2.2, YARN is part of the Hadoop project, and as such it is virtualization aware through the use of the Hadoop Virtualization Extensions. Like Mesos, YARN does not obtain its own resources. Adding machines to a YARN setup requires starting a YARN service on every machine. Nebu differs from YARN in that it obtains its own resources from a virtual machine manager. In addition, Nebu implements virtualization awareness by retrieving information about the physical topology it is running on. YARN requires an external system to provide the topology information.

4

Research, Design, and Development Processes

4.1. Overview

This section describes the process of Nebu's development, which can be split in three main categories. First, Section 4.2 describes some of the methods applied throughout the entirety of the project, such as the application of the Scrum methodology. Second, Section 4.3 describes the research processes utilised throughout this project, such as a survey on the use cases for the system through interviews. Third, Section 4.4 describes the process of designing for the system, focusing on the design process of the API. In addition, this section details some of the requirements that were changed during the course of this project and how these changes were handled by the team. Fourth, Section 4.5 outlines the different processes that concern development, such as the tools used to plan the individual sprints, during the implementation phase of this project and the tools used to ensure that all team members had a good overview of all the code that was written. Finally Section 4.6 reflects on the processes used and lists new insights gained from using these processes.

4.2. Project Management Processes

Throughout this project the team has applied the Scrum methodology, wherein teams work in intervals, known as sprints, with deliverables finished after every interval. This model of deliverables contrasts sharply with the more traditional waterfall method wherein different stages of the product cycle can be clearly defined. A project using the waterfall methodology starts with a design phase wherein the whole system is designed, followed by a production phase and ending with a validation phase. Scrum, as a form of agile development, advocates the use of short cycles consisting of these three phases instead. This is visually summarised in Figure 4.1. When using Scrum validation plays a more prominent role in the production process, ensuring that the final product matches the customers expectations.

For agile development to handle the frequent changes that are associated with it, communication is an essential part of the process. Three different types of meetings occurred on a frequent basis. First, the team met on a daily basis to inform one another about their current tasks and discuss what other tasks needed to be picked up. Second, meetings with the external supervisor took place every week to plan the next sprint in accordance with requests from the supervisor and issues as noted by the team. Finally, meetings with the TU coach were more irregular. These meetings primarily took place during the first and last three weeks of the project, discussing the orientation phase and the experimental work and reports mostly.

The main timeline of this project can be summarised in a total of eight sprints, which are listed in Table 4.1. Because the number of different tasks during the first two weeks was low, these weeks were merged into one sprint. During this sprint, the team focused on orientation. Additionally, the team conducted user studies to get a better understanding of the scope of the project. These user studies are described in Section 4.3.1.

The second sprint started the design and development process. This sprint was also used for the

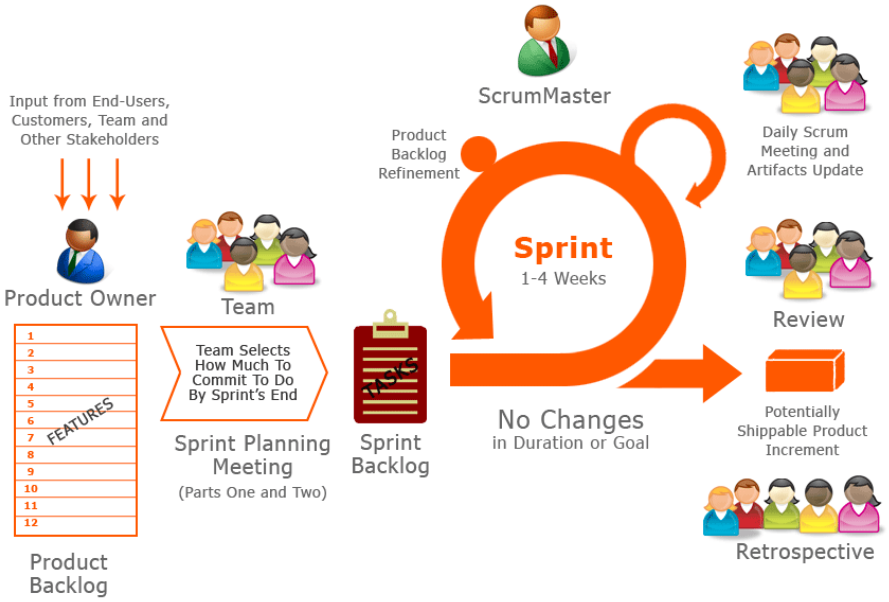


Figure 4.1: The Scrum methodology visually represented.

setup of tools like Jenkins. Sprints 4 through 6 consisted of design and development iterations. Sprint 7 moved the focus to the evaluation of both code and design. This resulted in removal of multiple code issues and cleaner code in general. Sprint 7 was also used to prepare for the experiments that took place in sprint 8. Sprint 8 was the last sprint and focused on conducting experiments and analysing the results. Finally, this sprint was also used to create the final deliverables.

By analysing Table 4.1, it becomes apparent that each of the sprints that focused on development can roughly be split in three main tasks. These are 1. development of the VMware extension, 2. development of the middleware, and 3. development of the application extension and system setup. These three tasks will be further described in Section 4.5, which describes how JIRA was used to plan work and how work was distributed among team members.

4.3. Research Processes

In preparation of this project an orientation of the current state of affairs was done on two levels. The first is a small study on the current processes Bitbrains uses in the operation of Hadoop and MongoDB in combination with VMware. This process, as well as the results, is detailed in Section 4.3.1. The second is a research survey in distributed systems, including some previous work, which is described in Section 4.3.2.

4.3.1. User Study

To establish a firm understanding of how Bitbrains currently manages its clusters and what their physical topology looks like, we conducted a small user study through the process of interviewing a number of Bitbrains employees. The employees, excluding our external supervisor, interviewed for this process are listed in Table 4.2. Each of these interviews was conducted in person and featured both specialised questions aimed at the expertise of the interviewee and some general questions related to broader concepts. In particular we interviewed people from a variety of categories. One visionary on a conceptual level, an engineer responsible for deploying clusters for customers, and an engineer responsible for the hardware and dealing with VMware. The results of these interviews have formed the basis of the design as presented in Chapter 5, and are further detailed in the orientation report that can be found in Appendix A.

Number	Week(s)	Main activities	Focus
1	1-2	Orientation in subject matter, survey, user studies.	Orientation, Research
2	3	Set-up Jenkins, get topology from VMware, pass topology through Nebu.	Design, Development
3	4	Convert topology to generic format, initial puppet-install, initial deployer.	Design, Development
4	5	Start VMs through VMware, query vSphere, start Nebu-common in C++.	Development
5	6	Add policies to the deployer, finish queries to vSphere, start MongoDB extension.	Design, Development
6	7	Policies support picking stores, vSphere offers stores, set-up DNS in the environment.	Design, Development
7	8	Add hooks for experiments, clean code for SIG review, create the Hadoop extension.	Evaluation, Development
8	9-10	Run experiments, write final report and prepare presentation.	Evaluation, Research

Table 4.1: Summary of the different sprints in the project, wherein week one corresponds to the week starting on the 21st of April 2014.

Name	Function description	Years of employment
Jeroen van Nieuwenhuizen	Engineer, Hadoop deployments	1
Bas Welman	Platform engineer, handles everything up to VM level	3
Gjalt van Rutten	CTO, deploying and implementing the technical vision	9

Table 4.2: Information about the employees interviewed for this project.

4.3.2. Research Survey in testing Distributed Systems

To determine the impact virtualization awareness has on characteristics such as performance and reliability, workloads to run on the distributed applications are chosen. Since no standards for testing distributed applications exist and real-world workloads are scarce, synthetic workloads such as TeraSort and WordCount are commonly used when benchmarking distributed systems. Previous work by Tim [5] includes a survey on various big data benchmarks and describes the BTWorld use case for time-based big data analytics. This work describes a complex real world workflow with a variety of MapReduce queries run on Hadoop. Unlike benchmarks such as TeraSort, BTWorld stresses multiple resources in a system and therefore provides a more complete assessment of the performance and reliability of a system. This workload is also run using the Nebu system, the results of which are described in Chapter 8, to determine what happens to the reliability of large complex workloads in real use cases.

4.4. Design Processes

Whereas the usage of the Scrum methodology prevents the creation of a detailed full system design at the start of the project, it does mandate the existence of evolving design documents. For Nebu most of the design can be captured in the public API exposed by the middleware and the VM manager extensions, which are described in Chapter 5. The usage of a modelling language called RAML for the evolving design documents of the APIs is described in Section 4.4.1. The API has been radically changed from a basic version to the current more powerful version due to some changes to the initial requirements. These changes and how they impacted our designs are described in Section 4.4.2.

4.4.1. Using RAML to handle Evolving APIs

To maintain a clear overview of the APIs offered by Nebu and the VMware extension, the RESTful API Modelling Language (RAML)[1] was used. RAML is built on YAML and JSON and offers a web interface where developers can specify, look up, and try multiple versions of their RESTful APIs. For this project both of the APIs have been fully documented in RAML, the first is the API exposed by Nebu to the

[1] http://raml.org/index.html

application, the second is the API that virtual machine managers are required to provide. We choose RAML to specify these RESTful APIs because of its ease-of-use and its ability to provide examples for each API call. Examples of this documentation are shown in Section 5.4.

Although the web-based editor for RAML is limited, it does allow for simple prototyping of APIs. This helped to quickly draft new additions to the APIs and to communicate changes among team members. After a first version of the API had been implemented, it became apparent that some changes in requirements meant this API would not be able to handle the requested operations. A new API could quickly be drafted, designed, and implemented through the use of RAML.

4.4.2. Changed Requirements

The changes in the design of the API were caused by changes in the requirements of the final product. Multiple requirements that were identified in the orientation phase of this project turned out to be unfeasible or unnecessary. This section describes why these changes occurred, how these changes were addressed, and how they impacted the design of the system.

No longer support Runtime Scheduling

Nebu's initial system requirements included the scheduling of VMs at runtime, allowing users or applications to specify restrictions on where VMs can be placed. It became apparent that in the case of VMware it is difficult to apply custom runtime scheduling. VMware will, through mechanisms of its own, continuously move virtual machines across hosts. If Nebu were to apply runtime scheduling on top of this, it would likely cause the two systems to clash, decreasing performance. Because the virtual machine manager has a complete view of the multi-cluster environment, its virtual machine scheduler is preferred over a custom scheduler located in the middle-ware layer. A possible work-around would entail Nebu advising the virtual machine manager's scheduler, rather than doing the scheduling itself. Initially, the Nebu design included such a mechanism. It would give advice to the scheduler about which machines to keep apart, and where to move virtual machines during a migration. However, the use-case for this feature at Bitbrains was limited and therefore dropped in favour of more advanced virtual machine deployment and placement.

Security Requirements

Nebu provides information about the physical topology to an engineer or customer. This information concerns the virtual machines that are under control of the engineer or customer that uses Nebu. However, a single Nebu instance can serve multiple users. This introduces a security issue, wherein users can access information that does not belong to their applications, unless some system of authorised access is introduced into Nebu.

This requirement was introduced during Nebu development. It was not identified as a requirement during Nebu's initial system design because it was not yet known where a Nebu instance would run and how many users it would be able to serve. Bitbrains informed the team that the result of this project should be a working prototype, rather than a production-ready application. As the implementation of such an authorisation system is a time-consuming operation that is not required for a working prototype, this requirement was assigned low priority and is not implemented in the prototype.

4.5. Development Processes

During the implementation phase of this project, the Scrum methodology was applied, with weekly sprints (as detailed in Table 4.1). Each Monday the team met with the external supervisor to plan the work for that week. The results of this planning would be inputted into *JIRA*[2], the process of which is further detailed in Section 4.5.1. JIRA is a project management and issue-tracking tool also used at Bitbrains. In addition to JIRA, which helped to enforce the Scrum methodology, the version control system git was used to keep track of the code. The way the team uses git is described in Section 4.5.2. Finally, the division of labour within the team is described in Section 4.5.3.

4.5.1. JIRA for Project and Issue Management

Engineers at Bitbrains use *Kanban*. Kanban is a software development workflow, in which tasks and user stories move through a pipeline of states. Examples of these states are "To Do", "In Progress",

[2]https://www.atlassian.com/software/jira

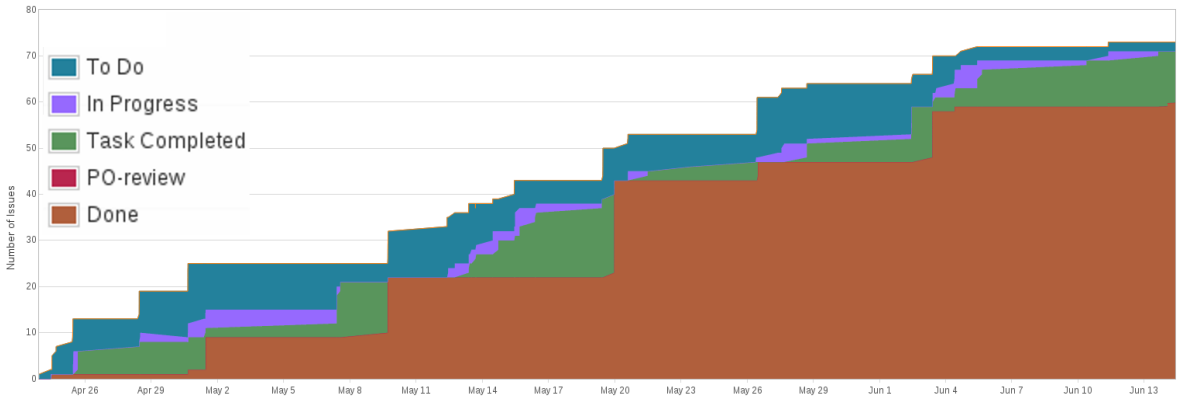


Figure 4.2: A cumulative flow diagram showing the different states tasks have gone through in JIRA. PO-review is not visible in the diagram because issues in this state were not tracked through JIRA.

"Task Completed", "PO-review", and "Done". The team complemented the use of Scrum with the use of the Kanban workflow. Since the team worked both at the university and at Bitbrains, it was not feasible to have a physical whiteboard with the tasks, as is common practice at Bitbrains. Instead the team opted to use a digital version of such an agile board, through use of JIRA.

At the start of every sprint, new tasks would be created in JIRA, split into smaller tasks that could then be done by one person. These tasks, similar to how Bitbrains works with the whiteboards, go through the states of the Kanban workflow. The accumulation of the tasks moving through these states is visually summarised in the flow diagram in Figure 4.2.

The PO-review (Project Owner review) was done at the end of every sprint by showing the new functionality to the supervisor and discussing its impact. Because this was done in an informal setting, tasks were never put in the PO-review state in JIRA. The spikes that are visible for the tasks in the Done state are caused by how the PO-review (not visible in the graph) was conducted, with all the completed tasks moving to the Done state at the end of every sprint. On average, the team finished nine tasks per week. This translates to three tasks per person, or roughly one and a half day for each task.

4.5.2. Version and Quality Control

Nebu uses git[3] for its version control. Git offers branching as a built-in feature. This allows for a main branch that contains the stable version of the software, and multiple other branches that contain new features that are being developed. The initial plan on how to use version control included a strict merge request policy with the goal to increase code quality. This policy meant that only code that had been reviewed by at least one other team member may be merged into the master branch. Other methods that were used to increase code quality included *SonarQube* and code evaluation by SIG. These methods are described in Section 7.3.1 and Section 7.3.2 respectively.

In practice, merge requests were only used when important features were introduced in the master branch. Smaller features could be silently added to the master branch as long as it breaks none of the available tests. Tests could always be silently added to the master branch. The main reason for deviating from the original plan was because the merge-requests proved to be very time consuming, and therefore not worth the effort for minor changes. The team felt that this time could be better spent developing. In addition, reducing the number of code reviews improved overall development speed.

4.5.3. Division of Labour

During the project each of the team members took on a major part of the project as identified in Section 4.2. Tim worked on the Nebu application extensions, Stefan worked on the Nebu-core project, and Jesse worked on the Nebu VMM extension for VMware. Additionally, Tim also performed the setup and management of external systems for both development and testing.

Besides these major tasks, each team member also took on smaller tasks on the side. These tasks

[3] http://git-scm.com/

were usually assigned to the person who either had the most time to spent on the task, or was more skilled in performing the task than the other members of the team. All team members also assisted other members with their tasks on a frequent basis. This was primarily done to speed up the completion of larger tasks.

4.6. Reflection

Although some deviations from the original plan have already been described in this chapter, this section focuses on the evaluation of the final method of execution of the described processes. Firstly, the Scrum methodology allows for more frequent feedback from the supervisor to keep the team on track. This made it easier to focus on the requirements that matter most to Bitbrains. It was through one of these feedback rounds that the supervisor mentioned Bitbrains prefers a properly working prototype with the core features over a system with many half-implemented features.

Something that could be improved upon was the use of code evaluations. In the current setup, merging code through the use of strict merge requests and code reviews prove to be too time consuming. For future projects, better tools could be used to accomplish efficient code reviews. Due to the relatively low number of code of reviews that were conducted during this project, it was hard for team members to make changes in code that was written by others. Code reviews would have ensured that at least two members had seen all of the code improving not only code quality, but also ensuring that at least one other member can extend the code if the original author is currently working on new functionality.

5

Design of the Nebu System

5.1. Overview

One of the main requirements of Nebu, as described in Section 2.2, is that it can support many distributed applications and VM-managers. To provide this support, Nebu solely communicates with other software through RESTful APIs. Nebu also uses a generic workflow, regardless of the application of VM-manager that is used.

To be compatible with multiple distributed applications and virtual machine managers, Nebu uses a modular design where it is completely independent of the distributed application and virtual machine manager that are used. This modular design is further described and motivated in Section 5.2.

Nebu can perform automated virtual machine deployment and placement at the request of the user. Section 5.3 gives a high-level description of how a user typically interacts with the system. This also includes retrieving physical topology information. A more detailed view of this interaction is given in Section 5.4, where Nebu's generic and efficient APIs are described.

Because there is no formal standard on how the physical topology of a multi-cluster environment should be organised, different multi-cluster environments can have different physical topologies to accommodate various workloads and applications. Nebu represents information about the physical topology of a multi-cluster infrastructure in a generic format, to be compatible with differences in topology across multi-cluster environments. This format is described in Section 5.5.

For the automated placement of virtual machines, Nebu uses placement policies. These placement policies provide mappings of the virtual machines to the physical host on which they are placed. Because selecting the best placement policy is non-trivial and depends on the distributed application, this is left up to the user. The different placement policies Nebu provides are described in Section 5.6.

MongoDB does not automatically use physical topology when this is provided. To allow MongoDB to use this information, a MongoDB replication policy is designed and discussed in Section 5.7.

5.2. System Modularity

Because Nebu needs to be compatible with multiple distributed applications and virtual machine managers, as described in Section 2.2, it uses a layered design. The objects in each layer are independent of those in other layers. The Nebu core forms the middle layer, between the application extension layer and the virtual machine manager layer. An overview of this design is shown in Figure 5.1. The multiple layers communicate through RESTful APIs. When adding support for a new distributed application, an extension needs to be added to the distributed application layer. This can happen without a system reboot, as new distributed applications can just start conversing with the RESTful API. Similarly, when support for a new VMM is added, an extension for this VMM needs to be added in the VMM layer. This, however, does require a system reboot as Nebu can currently only talk to one virtual machine manager at any given point in time.

The application extension communicates with the the Nebu core and is responsible for making informed decisions for the application based on the information retrieved from the core.

The Nebu core provides an interface to both users and applications. The core has two main responsibilities. First, it provides information about the physical topology of the multi-cluster infrastructure

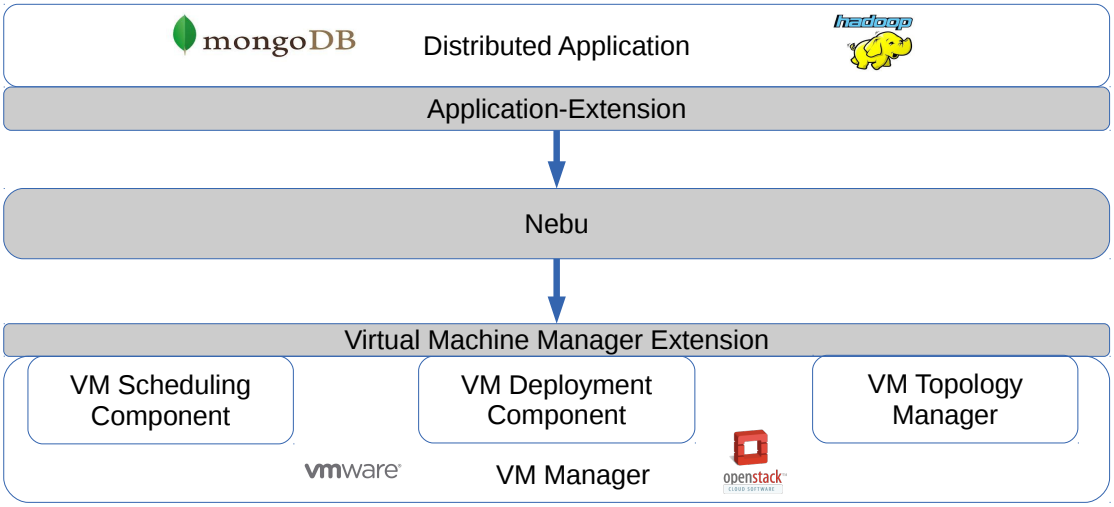


Figure 5.1: The Nebu system, indicating the different layers and components required in the system, communicating through RESTful APIs. The components in grey indicate components of the Nebu system.

on which the virtual machines are deployed. Second, it provides an interface through which users can deploy distributed applications using topology-aware placement policies.

The virtual machine manager (VMM) extension communicates directly with the VMM that is being used for the deployment of virtual machines. It retrieves information about the virtual machines and physical resources from the multi-cluster infrastructure. Examples of physical resources include physical hosts, racks, data centres.

5.3. User-System Interaction

Any Nebu user is able to perform an automated deployment of a distributed application in a virtual environment as specified by requirement 4 and 5, see Section 2.2. Because this could be any application or virtual environment Nebu uses a generic workflow, regardless of the application or VM-manager that is being used. To perform a deployment, a fixed sequence of steps need to be executed.

1. The user should create a new Nebu 'app'. This forms a reference to the application the user would like to deploy.
2. The user should inform Nebu what kind of VMs his app will use. This information consists of two parts. The first part is the general information that Nebu will use to provide a deployment. This information includes a name for the type of virtual machine, and an estimate of how resources such as CPU, memory, disk and network are used by this VM. The second part is VMM-specific information containing details for the virtual machine manager that is not relevant for the Nebu core. The VMM extension that is used is free to specify the structure of this data.
3. The user specifies a placement policy and requests a deployment specification. This specification creates a mapping of virtual machines to physical hosts. The specification is presented to the user to allow them to make modifications if this is desirable.
4. The user starts the automated deployment based on the deployment specification. After this, the deployment is complete and no further action is required.

Each deployment needs its own app in Nebu. This allows Nebu to perform multiple deployments of multiple applications. The user does not have to specify what distributed application they want to deploy. Instead, the user should start the corresponding application extension and provide it with the app ID from Nebu core.

This approach has two main benefits. First, starting a new application extension process per deployment makes scaling easy because each application extension only manages one application deployment, and there is no system limit on how many application extensions can be started. Second,

the possibility to run Nebu core separately from the application extensions increases system security. The Nebu core does not require access to the VMs it deployed. Only the application extension needs access to these machines. This allows multiple users to access Nebu core without giving these users access to each others virtual machines.

5.4. Design of a RESTful API

Nebu operates as middleware between the VM manager and the distributed application. To be able to communicate to both sides, Nebu requires two RESTful APIs to be available. The first API is offered by Nebu to the application extension. The second should be provided by the VM manager extension to Nebu. These APIs and how they satisfy the requirements specified in Section 2.2 are described in sections 5.4.1 and 5.4.2 respectively. For details about specific API calls please refer to Appendix E.

5.4.1. Nebu RESTful API

The Nebu RESTful API provides an interface to both distributed applications and the engineers that manage these applications. Its main features are:

1. Generating deployment specifications.
2. Performing VM deployment and placement.
3. Providing information about the physical topology of the cloud to the application.

The Nebu API is an efficient API that allows the user to perform an automated deployment with minimal effort compared to current techniques used by Bitbrains engineers. Each of the four major steps described in Section 5.3 can be completed with a single API call. This means that a user that performs all steps required for a complete distributed application deployment manually, which can take half a day or more, can achieve the same result by calling just four Nebu functions.

Requirement 5 states that the placement of virtual machines should be configurable. However, assigning each virtual machine to a physical host manually is labour intensive and defeats the purpose of using Nebu to do the heavy lifting. To help users be more productive, Nebu allows users to specify placement policies. These policies are predefined algorithms that Nebu can use to calculate a virtual machine to physical host mapping. Such a mapping is called a *deployment specification*. The policies Nebu uses to calculate these deployment specifications are described in Section 5.6. To keep users in control, they can manually edit the deployment specification to their specific needs.

When the user is satisfied with a deployment specification, they can trigger a virtual machine deployment. This deployment starts new virtual machines and places them according to the deployment specification. This process is completed without further interaction from the user. This meets requirements 4 and 5 from the system requirements.

Finally, the API enables applications to ask for the physical topology of the cloud infrastructure. This call is performed on a per-application basis. This allows the API to hide physical topology information that is not relevant to the application. As an example, consider a multi-cluster infrastructure consisting of two data centres. If an application can only be deployed in one of these data centres, the physical topology of the other data centre is irrelevant. Providing physical topology information to applications satisfied requirement 6.

Because this API is a RESTful API, it is easily extensible for distributed applications. The only requirement is that at least one of the machines that run the distributed application can establish a connection to the machine that runs Nebu. This satisfies requirement 2. An overview of the API is shown in Figure 5.2.

5.4.2. VM Manager RESTful API

The VM Manager RESTful API provides an interface to the Nebu middleware. Its main features are:

1. Deployment and placement of virtual machines.
2. Providing physical topology information.
3. Providing virtual machine information.

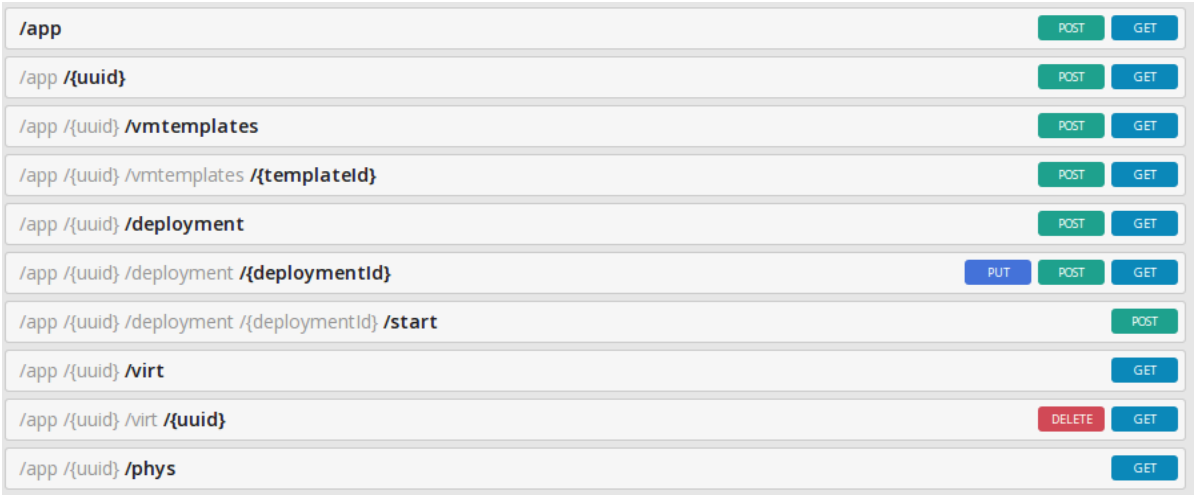


Figure 5.2: The Nebu core RESTful API.

Each supported virtual machine manager must provide this RESTful API to the Nebu core. The overview of the API is shown in Figure 5.3.

For Nebu to provide information about the psychical topology of the cloud infrastructure and perform automated virtual machine deployment, it needs to communicate with a virtual machine hypervisor. However, to make Nebu compatible with multiple hypervisors as indicated by requirement 3, the API should be generic and minimal.

Satisfying requirements 4 and 5 require virtual machines to be deployed and placed on specific physical hosts, respectively. Because a Nebu deployment specification always specifies on which physical host a VM should be placed, the API call to create a new virtual machine combines these two functionalities. When the Nebu-core requests a new virtual machine to be deployed, it must specify the physical host where the VM should be placed. This makes the API more efficient without losing any functionality.

The VM hypervisor API enables Nebu to retrieve the complete physical topology of the multi-cluster infrastructure. Unlike the API of the Nebu-core, this API retrieves the complete topology. The primary reason for this behaviour is that Nebu can support multiple applications at the same time. These applications might require different topology information. In this design, this information can be retrieved in a single call to the hypervisor extension, in stead of several separate calls. This information is then cached by the Nebu-core. This reduces the load on the hypervisor extension. Retrieving physical topology information is specified by requirement 6.

To allow the Nebu-core to manage its application deployments, it needs to be able to retrieve information about the virtual machines it has deployed. The hypervisor extension provides this functionality by offering an overview of all virtual machines in the multi-cluster, or detailed information about individual virtual machines. The administration that specifies which VM belongs to which application is done by the Nebu-core. This reduces the load on the hypervisor extension.

5.5. Physical Topology Abstraction

Different cloud providers are likely to have different multi-cluster infrastructures. In addition, different virtual machine managers will likely use different classes to represent these topologies. Nebu is compatible with multiple VM-managers. It is therefore important for Nebu to be able to convert a VM-managers specific topology representation to a generic representation that can be used throughout Nebu. This section describes the format chosen to represent the topology and argues why it satisfies requirement 1.

To model the physical topology a tree-like structure has been designed starting with a root or "cloud" object representing the multi-cluster infrastructure as a whole. On the first level we find data centres, which represent the next largest entity in the physical topology. In case of the Bitbrains topology, three such data centres exist. Within a data centre the topology houses rack entities. These in turn

Figure 5.3: The Nebu VMM RESTful API.

can contain stores and hosts. The stores contained in racks represent the network attached storage in the topology, whereas the host entity can also contain stores to represent local storage. All of this is visually represented in Figure 5.4. Each node in this topology forms a *failure domain*. This means that when a node fails, only the children of that element will be affected by the failure. As an example, consider a rack under heavy load. The load can strain the racks power supply and cause instability. When this happens, only the nodes that are part of this rack are affected.

This tree like structure is very similar to other commonly used models, such as the model adopted by Hadoop. Hadoop's model also features data centres, racks and nodes, but does not distinguish between different nodes. The nodes in Hadoop are all generic data nodes, whereas the model described above allows for a distinction in hosts, network storage and local storage.

The model can easily be mapped to the physical topology that is in place at Bitbrains, since they currently use three different data centres, wherein they have clusters containing blade servers and storage units. These clusters can be mapped to the racks, the blades to hosts and the storage to the applicable node type. The model is generic enough to cover other topologies as well, An example is the DAS-4[1]. The DAS-4 is a distributed supercomputer shared among Dutch universities and hosts a variety of nodes. Each of the universities hosts its own site which can be mapped to data centres in this model. Each of these sites consists of multiple racks that contain a variety of physical machines. These can be mapped to racks and hosts respectively.

5.6. Placement Policies

One of Nebu's main features is the placement of virtual machines onto physical hosts to improve the performance and reliability of distributed applications. The deployer component bases its decisions on the selected placement policy, of which Nebu currently supports four. These policies allow users to configure the VM placement and reach some or multiple of these goals. The use of placement policies is part of requirement 5. In this section we will further describe the different policies in terms of their goals, their algorithms and their intuition.

5.6.1. The Random Policy

Intuition

The random policy provides a base line for the experimental work described in Chapter 8, since a random policy does not offer any guarantees on performance or replication. In addition it is comparable to how

[1] http://www.cs.vu.nl/das4/home.shtml

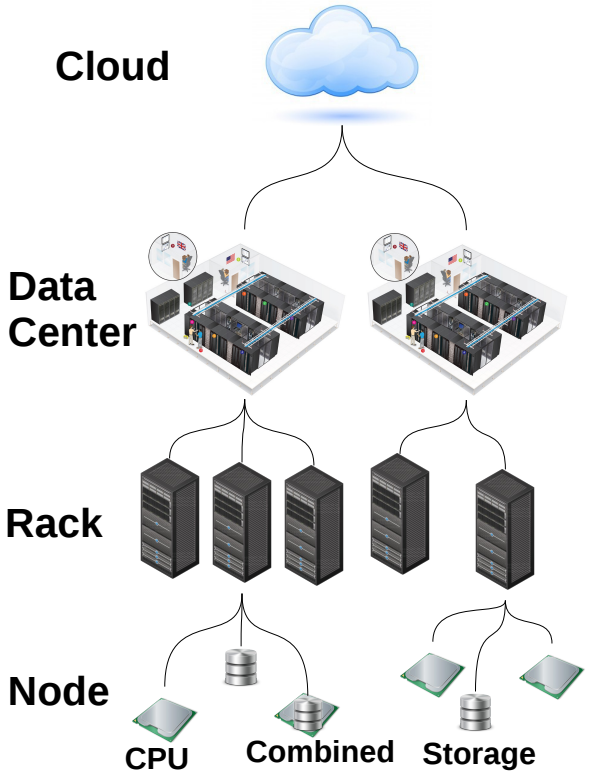


Figure 5.4: Model of the topology as designed for the Nebu system.

a client might get virtual machines from commercial cloud providers such as Amazon EC2, which also offers no guarantees on the placement of virtual machines.

Algorithm

The algorithm for this policy is by far the simplest and has been described in pseudocode in Algorithm 1. As can be seen, no stores are selected for the virtual machines. When deploying a virtual machine without selecting a store, a random disk is selected by default.

Algorithm 1 Random Policy.

```
Input:
  numVMs ← the number of VMs that need to be deployed.
  hosts ← all the hosts available for deployment.
Output:
  chosenHosts ← the hosts chosen to deploy the VMs on.
  function deploy
    for i ← 1 ... numVMs do
      index ←random() % size(hosts)
      chosenHosts[i] ← hosts[index]
    end for
  end function
```

5.6.2. The Locality Policy

Intuition

Reduction of network communication overhead, either by reducing the amount of traffic, or increasing the throughput, can increase the performance of distributed applications. Virtual machine managers

such as VMware offer certain mechanism to assist in the reduction of network communication. The locality policy tries to place all the virtual machines as closely together as possible, even placing them on the same hosts. In doing so, VMware can use in-memory transfer to transfer data between two virtual machines on the same host. Although this policy can have a negative effect on an application's reliability, it probably improves performance.

Algorithm

The algorithm for this policy first determines where the other virtual machines of the current application have already been launched (if any), before selecting the most used rack first. It then sorts the hosts in that rack by number of virtual machines on them, filling them up with extra virtual machines as it goes along. The policy also takes a configurable parameter in the form of the maximum number of virtual machines that can be placed on a single host. The algorithm is described in pseudocode in Algorithm 2.

Algorithm 2 Locality Policy.

```
Input:
  numVMs ← the number of VMs that need to be deployed.
  allRacks ← all the racks available for deployment.
Output:
  chosenHosts ← the hosts chosen to deploy the VMs on.
  function deploy
      racks ← sort allRacks by VMs on rack, most VMs first
      while numVMs > 0 do
          rack ← racks.next()
          hosts ← sort rack.getHosts() by VMs on host, most VMs first
          while numVMs > 0 do
              host←hosts.next()
              while numVms > 0 and host.getNumVMs() < maxVmsPerHost do
                  chosenHosts ← chosenHosts ∪ {host}
                  NumVms ← NumVms −1
              end while
          end while
      end while
  end function
```

5.6.3. The Replication Policy

Intuition

The complete opposite of the locality policy is represented by the replication policy. Instead of placing the virtual machines as close together as possible, it tries to divide the virtual machines as much as possible. The policy is configured to spread the virtual machines on a rack-level, always choosing the rack with the least virtual machines on it. In doing so, an attempt is made to change the smallest point of failure from that of a physical host to that of a whole rack, an event that is far less likely to happen.

Algorithm

The algorithm for this policy is best described as choosing the rack with the least virtual machines, followed by choosing the host with the least virtual machines. This is summarised in pseudocode in Algorithm 3.

5.6.4. The Local-Remote Policy

Intuition

This final policy is inspired by the way Hadoop distributes its replicas and has been implemented to synchronise well with this manner of distribution. When a mapper in Hadoop has to write a new block of data it places the first copy on its own local host, but writes two duplicates to two remote hosts that are close together. For instance, in a multi-cluster environment, a work node will place one copy locally

Algorithm 3 Replication Policy.

Input:
numVMs ← the number of VMs that need to be deployed.
allRacks ← all the racks available for deployment.
Output:
chosenHosts ← the hosts chosen to deploy the VMs on.
function deploy
while numVMs > 0 **do**
rack ← least used rack from allRacks
host ← least used host in rack
chosenHosts ← chosenHosts ∪ {host}
NumVms ← NumVms -1
end while
end function

and place two copies on another data centre. To facilitate this, the local-remote policy tries to create a deployment in which two third of the virtual machines are in the same rack, and one third of the virtual machines are in other remote racks. A virtualization aware Hadoop can profit from this, since this distribution of virtual machines offers both reliability guarantees due to the spread over different racks/data centres and a possible performance increase since two thirds of the machines are located closely together.

Algorithm

The algorithm for this policy starts by calculating how many virtual machines it needs to add to the most used rack, which is the rack that will contain the two thirds of the total number of virtual machines. The other virtual machines will be added to other random racks. This algorithm is described in pseudocode in Algorithm 4.

Algorithm 4 Local-Remote Policy.

Input:
numVMs ← the number of VMs that need to be deployed.
allRacks ← all the racks available for deployment.
Output:
chosenHosts ← the hosts chosen to deploy the VMs on.
function deploy
twoThirds ← $\frac{2}{3}\cdot$ numVms
fatRack ← most used rack from allRacks
for $i \leftarrow 1 \ldots$ twoThirds **do**
chosenHosts ← chosenHosts ∪ {least used host from fatRack}
end for
for $i \leftarrow 1 \ldots$ numVms - twoThirds **do**
host ← least used host not in fatRack
chosenHosts ← chosenHosts ∪ {host}
end for
end function

5.7. MongoDB Replication Policy

The Nebu extension for MongoDB is responsible for grouping available virtual machines into sets of three to form replica sets. To achieve this, a policy for forming replica sets is present in the implementation of the MongoDB extension. The policy is generic in that it can be used to deploy any distributed data store that uses replication and allows external software to define the location of replicas.

Intuition

The Nebu MongoDB replication policy has at its core the goal that no two VMs in a replica set should be on the same host or disk. It also uses several intuitive claims about the selection of VMs for a replica set and how that selection impacts the possibilities for forming other replica sets thereafter. First, whenever one third of the available virtual machines resides on a single rack, at least one of those VMs must be included in the next replica set. This ensures that the fraction of VMs residing on a single rack does not increase further. When not taking this into account, later replica sets may need to be placed within a single rack, which increases the chance of them being on the same host or store. Second, by choosing virtual machines that share hosts or stores with many other virtual machines, the chances of having collisions in future replica sets is reduced.

Algorithm

Algorithm 5 provides pseudocode for how the MongoDB replication policy forms a single replica set, given a list of available virtual machines and a list of racks. The algorithm first determines for each rack the minimum amount of VMs that must be placed on it. Then it loops for all 3-combinations of virtual machines and determines for each combination whether it complies with the main goal and the first intuitive claim mentioned in the Intuition. In addition, it assign a "value" to each combination that does comply based of the amount of future collisions it prevents (as per the second intuition). After all combinations are evaluated, the combination of VMs with the highest value is chosen to become a replica set.

Due to the complexity of the input of this algorithm, no complete analysis has been done on the optimality of the output. As this policy is used only for a prototype extension for MongoDB, no time was spent to provide this analysis. In practice, this algorithm has not failed to form distinct replica sets where they were possible. For a production-ready implementation, a detailed analysis will be required.

Algorithm 5 Replication policy used in the Nebu MongoDB extension.

```
Input:
  vmList ← list of virtual machines not part of replica set.
  racks ← list of racks in the physical topology.
Output:
  replSet ← replica set containing three VMs.
  function formReplicaSet
    oneThirds ← size(vmList) / 3
    for all racks r do
      vmsOnRack[r] ← amount of vms in vmList residing on rack r
      minOnRack[r] ← floor(vmsOnRack[r] / oneThirds)
    end for
    for all 3-combinations V in vmList do
      if V contains duplicate hosts or stores then
        value[V] ← 0
      else if V does not meet minimum no. VMs per rack then
        value[V] ← 0
      else
        value[V] ← 1 + (no. VMs in vmList sharing a host or store with a VM in V)
      end if
    end for
    replSet ← 3-combination V with highest value
  end function
```

6

Development of the Nebu System

6.1. Overview

Nebu is a complex software system that consists of seven independent software projects. Together, these projects span almost ten thousand lines of code. Additionally, many of the Nebu projects interface with complex external systems such as Hadoop, MongoDB and VMware. Finally, Nebu must be compatible with enterprise environments, which significantly restricts the technologies that can be used by the system. For these reasons, the development of the system a challenging task.

In this chapter all parts of the system are described in detail. Section 6.2 describes the different projects in the Nebu system and the relation of each part of the system with the system requirements, which are described in Section 2.2. An overview of all system requirements is shown in Table 2.1. The different external libraries and development tools used by and for Nebu are described in Section 6.3. The technical difficulties encountered during the development are described in Section 6.4.

6.2. Product Description

For this thesis a total of seven projects have been implemented, three of which are written in the Java programming language and four in C++. One of the projects, Nebu-common, is implemented in both languages. The Nebu-common project provides the generic physical topology representation described in Section 5.5. This representation meets requirement 1. Additionally, it increases the usability of the product by making the topology representation available to both Java and C++ programmers. Nebu-common is described in more detail in Section 6.2.1.

The other two Java projects are Nebu-core, and the Nebu VMM extension for VMware. These projects are described in detail in Section 6.2.2 and Section 6.2.3 respectively. The Nebu-core project provides an implementation of the extensible and efficient Nebu RESTful API. This API can be queried by distributed applications and external users. It is described in detail in Section 5.4.1. This API meets requirement 2. The Nebu VMM extension for VMware communicates directly with VMware and provides an implementation of the extensible and efficient VM Manager RESTful API. This API is described in Section 5.4.2. This API meets requirement 3. Additionally, this extension offers one of two extensions specified in requirement 10.

When the Nebu-core is combined with a VMM extension, it is capable of providing distributed applications with information about the physical topology of the cloud infrastructure, and inform these applications where their virtual machines are located. It is also capable of performing automated virtual machine deployments and virtual machine placements. These three capabilities meet requirements 6, 4, and 5 respectively.

The Nebu application extensions for Hadoop and MongoDB are implemented in C++. These projects provide ways to answer the second research question as stated in Section 1.2. This validation is done empirically through experiments, which are described in Chapter 8. These extensions also increase the usability of the system and meet requirement 9. These extensions are described in detail in Section 6.2.4. To increase system usability even more, a C++ application-framework is provided which makes it easier for engineers to create an extension for other distributed applications in the future.

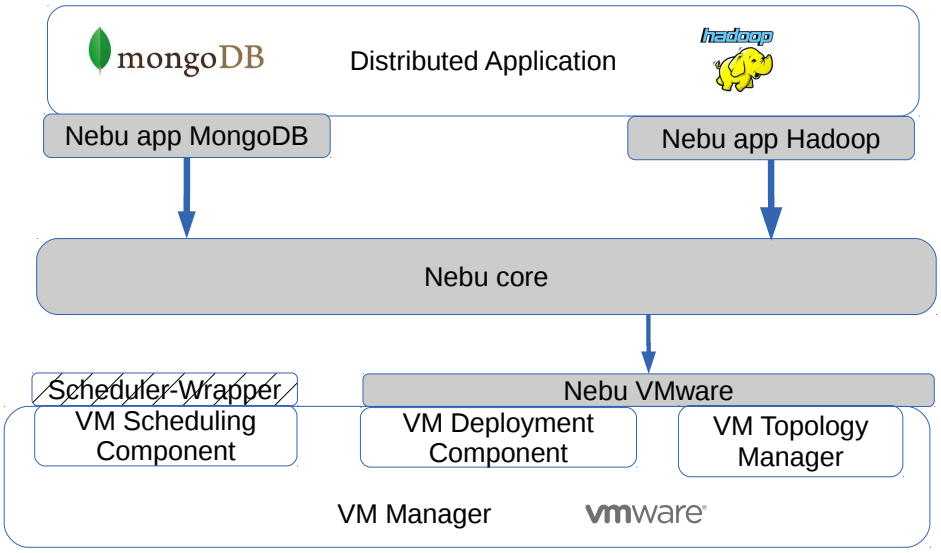


Figure 6.1: The Nebu system showing where the different projects are placed in grey.

The locations of the different projects in the software ecosystem are visually represented in Figure 6.1. As the figure depicts, the Nebu VMware project handles communication with both the deployer as well as the topology section of VMware, as the design prescribes all operations should be available in one API. In addition the scheduler wrapper has been marked as not implemented in accordance with that which is described in Section 4.4.2. In addition OpenStack has been removed from the figure, as no extension of this was created. For more information see Section 6.4.5.

6.2.1. Nebu Common

Examining the software ecosystem in Figure 6.1, it becomes clear that some of the data needs to pass multiple layers from its source to its destination. For instance, topology information is provided by the VM-manager extension, processed and passed along by Nebu-core and finally used by the distributed application. The different classes that hold information on items such as physical hosts, physical stores and virtual machines have therefore been placed in a library, called Nebu-common that has been implemented in both C++ and Java.

Overview

The class diagram depicted in Figure 6.2 shows the different packages and the classes contained in them that have been created in the Nebu-common library. It should be noted that this diagram represents the Java version of the library. Whereas the C++ version offers similar classes (at least for the topology related classes), not all code has been ported due to time constraints and lack of necessity for this prototype.

An important package in this library is the interfaces package, which holds only two small interfaces. The first is the `Identifiable` interface, which mandates a class to have a `getUniqueIdentifier()` method that returns an ID of the object. Many classes, both in Nebu-common, -core, and -VMware implement this interface for easy identification of an object. Similarly the `IBuilder` interface is used to implement the builder pattern from software engineering. This pattern prescribes the usage of builder classes, which can be configured by using `withX()` calls to pass parameters to the object you want to build. This method allows you to use default values where none are specified, but also make some parameters for a class mandatory, without either having many different constructors or having constructors that can throw exceptions. These violate readability and general code standards within software engineering respectively.

The second most important and most used interface is that of the `XMLFactory` which is in the util.xml package. This interface prescribes the implementation of a `toXML()` and `fromXML()` method to allow objects to be transformed into the XML-format that can be transmitted over the HTTP protocol

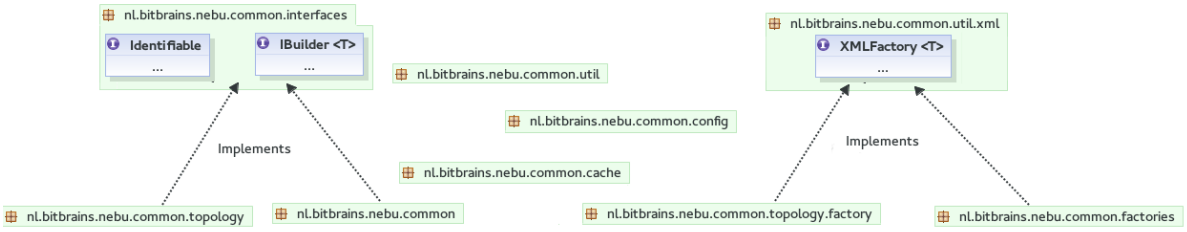


Figure 6.2: Package diagram of the packages and classes in Nebu-common.

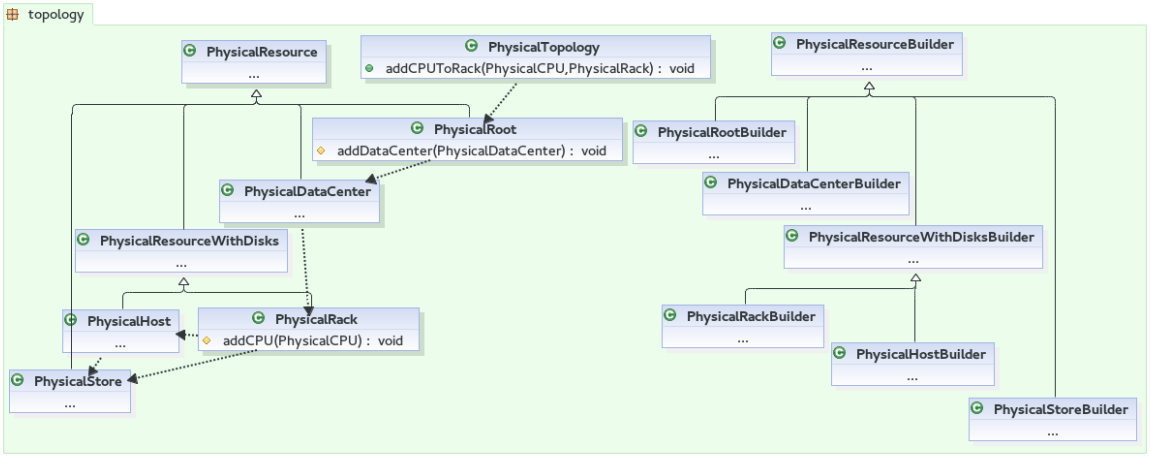


Figure 6.3: Class diagram of the topology classes in Nebu-common.

and is still human-readable. The topology package which contains classes that can hold information on the physical topology of the network is the largest one and will be described in more detail below.

The `cache` package implements a very simple in-memory caching mechanic for REST-calls or other central storage of information in the application. If Nebu required data persistence, this class can simply be extended to communicate with a database in which it can store its information (see Section 9.2.2). Since this was no system requirement during this project, this feature has not been implemented yet.

The main package of this project `nebu.common` holds the `VirtualMachine` class, which holds information on a virtual machine. The `config` package holds some simple containers for configuration that can be read from XML to configure for instance the port an application should connect to or listen on (something both Nebu-core and Nebu-VMware require), so that the parts of Nebu could be run on different machines, and finally the `util` package holds some common static functions that for instance perform null checks on arguments.

The Topology Information Containers

By far the largest package is the `topology` package, combined with the child package `topology.factories`. These two packages are an implementation of the generic physical topology representation described in Section 5.5. As the class diagram in Figure 6.3 illustrates, these classes are all relatively simple containers. The builders for these classes have also been included in this more detailed class diagram, to clearly show how the builder pattern has been implemented in these container classes. In this analysis of the package however, only the actual data containers will be further described, as the builders only serve as a way of constructing the actual containers.

Instead a `PhysicalTopology` class has been created that holds a PhysicalRoot and offers all of the

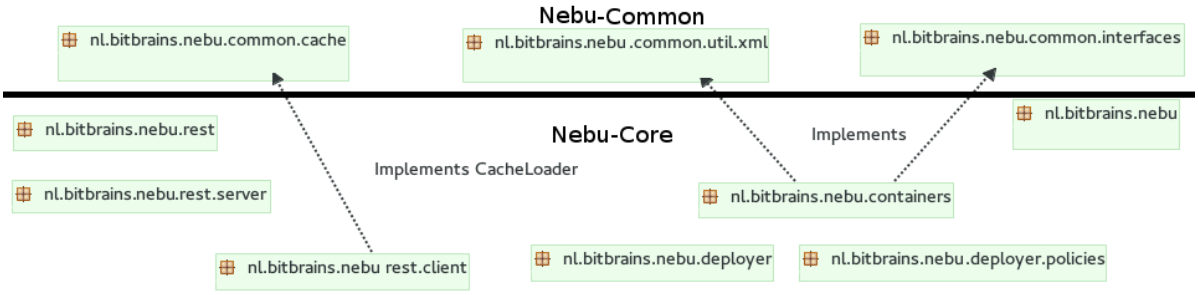


Figure 6.4: Package diagram of the packages and classes in Nebu-core.

topology altering operations (such as inserting or deleting `PhysicalHost` objects). The philosophy behind this decision is that it can be guaranteed that links between two objects are always two-way, i.e. if an object x believes y to be its parent, then y believes x to be its child. By making the topology altering methods of the containers public, it is possible to set a parent of an object, without adding the object to that parent as a child. In addition, the `PhysicalTopology` class contains some convenience methods such as `getCPUByID()` to prevent manually crawling the entire topology every time you need a specific host.

6.2.2. Nebu Core

The largest of all projects in Nebu is, as the name hints, Nebu-core. This project is located between the application and VMM extensions and communicates with both. Additionally, it calculates new deployment specifications and manages existing deployments.

Overview

The package diagram of Nebu-core in Figure 6.4 shows the packages in this project and how they relate to one another. The `rest.server` package holds the classes responsible for the exposed API, which will be discussed in more detail below. The deployer package and the corresponding `deployer.policies` package hold implementations of the policies described in Section 5.6.

The `containers` package of Nebu-core holds the main data structures behind the API, namely applications, vmtemplates and deployments which represent the three entities that can be queried through the API. All four of these containers use a builder pattern and have XML factories to convert them to and from XML for use in the API, hence the relation with the `util.xml` and `interfaces` packages from Nebu-common.

Another main feature of Nebu-core, which is communication with the virtual machine manager API is done by the `rest.client` package. A central class which uses the Singleton pattern is used to perform the actual requests, which in turn provide CacheLoaders. These CacheLoaders are responsible for saving the information obtained from the virtual machine manager API in the cache.

The implementation of the API

Nebu-core has been designed to allow easy extension in the future. Two packages are particularly likely to be extended when new functionality is added. One being the `deployer.policies` package as new policies are added to the system and the other being the `rest` package. As the policies are mapped from the pseudocode presented in Section 5.6 to the Java classes shown in the package diagram, we will focus here on the `rest` package and more specifically the `rest.server` package. If more API calls need to be supported by Nebu, this is where an extension would have to be made.

To provide the API as specified in Section 5.4, several different provider classes have been created. These classes are shown in the class diagram of Figure 6.5. The way this is set-up, has every class be responsible for one part of the URI. For instance, the `AppsProvider` class handles the `/app` part of the URI, whereas the `AppProvider` class handles the `appID` part. A full example of how the URI parsed is also shown in the figure, indicated by the coloured arrows. By ensuring that each class handles only one part of the URI, we not only adhere to the Single-Responsibility principle from software engineering, but also ensure that error checking in the URI can be centralised. For instance

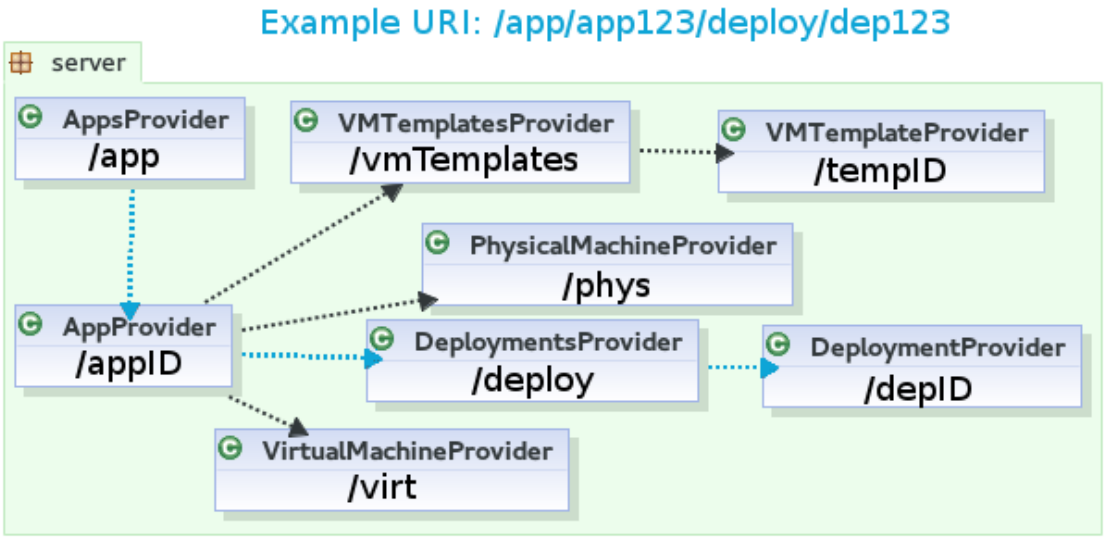


Figure 6.5: Class diagram of the provider classes in Nebu-core, wherein the blue arrows indicate how the example URI is parsed.

returning an HTTP 404 code for an invalid application id can now be handled in the `AppProvider`, rather than the check being performed in all of the other providers.

Each provider is given the data structures it requires to return the requested information or perform the requests, but has no access to any other information. This was designed and implemented as such with authorisation-regulated access in mind (see Section 9.2.1).

As an example of how HTTP requests are processed by the Nebu-core, a sequence diagram illustrating the methods that are called internally is presented in Figure 6.6. For the sake of readability, the methods are displayed as being called directly on the required provider. In reality however, all of these calls go through AppsProvider and then go to AppProvider etc. as required for the call to be handled.

6.2.3. Nebu VMM Extension

The Nebu-VMM extension for VMware is a project that provides Nebu compatibility for VMware. It is built on top of vCloud and vSphere and offers the RESTful API from Section 5.4.2 to the Nebu-core.

vCloud is a VMware product that allows users to manage private virtual machines in a commercial cloud. It allows users to manage virtual machines without giving the user access to the physical hardware. Among other things, it is capable of deploying virtual machines, and setting up virtual resources such as virtual networks and virtual storage devices. vCloud itself is built on top of vSphere.

vSphere is a management tool for a VMware cloud. It allows users to manage the complete VMware infrastructure from the physical hardware to the virtual machines. Among other things, it provides information about the physical topology of the multi-cluster environment in manages. vSphere features a permission system that allows different users to view and modify different parts of the environment. The vSphere permissions that are required to successfully run Nebu are shown in Table 6.1.

Overview

This section provides an overview of the project and describes the packages that together form the Nebu VMware extension. A complete overview of the packages is shown in Figure 6.7. Because this extension was written in Java, like the Nebu-common project, it can borrow some of its functionalities. The `VMTemplate`, `VmBootStatus` and `VirtualApplication` classes inherit the `Identifiable` interface from Nebu-common. This interface ensures that objects from these classes are always uniquely identifiable.

The `converter` package is responsible for translating objects from one representation to another. More specifically, it is responsible for translating objects that are returned by the VMware APIs to objects from Nebu-common. This includes virtual objects such as virtual machines, and physical resources such as physical hosts, racks, data centres, and storage units.

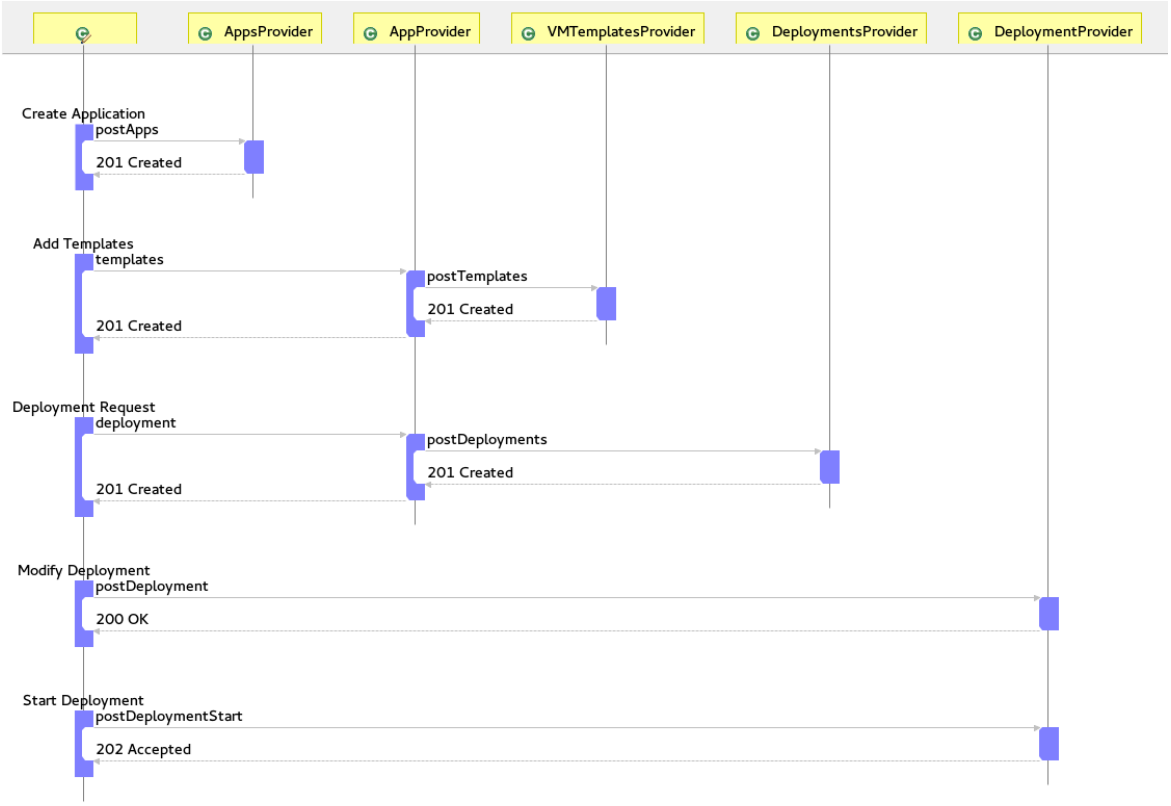


Figure 6.6: Sequence diagram of the provider classes in Nebu-core.

The provider package consists of the classes that provide the RESTful API. These classes correspond with the four major parts of the VMM API as specified in Section 5.4.2. For example, the `VMTemplateProvider` is responsible for all calls on `/vmtemplates`, the `VirtualResourceProvider` is responsible for all calls made on `/virt`, etc.

VMware interface

The VMware extension uses APIs from vCloud and vSphere. Interactions with these APIs go through the vcloud and vsphere package respectively. Which exact VMware APIs are used internally does not matter for the external functionality and are susceptible to change. This could happen when a better API becomes available, or one of the VMware APIs becomes deprecated. For this reason, the VMware extension code does not directly interact with the vcloud and vsphere packages. In stead, this part of the project is hidden behind an interface called 'VMware'.

The VMware interface hides all calls to external APIs that are used. It defines the generic calls that should be handled by VMware, regardless of the exact API that is used to handle these calls. The implementation of the VMware interface is called 'DefaultVMware'. It uses vCloud and vSphere to handle calls made to VMware.

Because the interface hides all external communication, many classes call this interface. An example is shown in the VMware class diagram shown in Figure 6.8. Many of the provider classes directly call methods from the VMware interface. All interface methods only return types that are defined either in Nebu-common or the VMware extension. The interface hides all objects from the external APIs. The class that implements this interface is responsible for converting these objects, which is done by using the converter package, described earlier in this section, described earlier in this section.

One of the calls made by the PhysicalTopologyProvider is 'createVM'. A sequence diagram of this call is shown in Figure 6.9. When a user wants to create a new virtual machine, they provide the UUID of the VM template they want to use for deployment, and the physical host where the VM should be placed. Objects that match these identifiers are retrieved through the VMware interface. These

Entity	Permission name
Datastore	Allocate space
Resource	Migrate powered off virtual machine
Resource	Migrate powered on virtual machine
Scheduled Task	Create tasks
Scheduled Task	Modify task
Scheduled Task	Remove task
Scheduled Task	Run task
Virtual Machine Interaction	Power off
Virtual Machine Interaction	Power on
Datastore cluster	Configure a datastore cluster

Table 6.1: vSphere permissions required to run Nebu.

are represented by the first two calls made by the PhysicalTopologyProvider. The third call shown in the diagram initiates the creation of a new virtual machine. The VirtualMachine and the PhysicalHost objects that were retrieved are passed as arguments. The deployment of the virtual machine is run in a separate thread. The method returns an object that allows asynchronous monitoring of the deployment and placement of the new virtual machines.

vCloud

The vcloud package is used by the DefaultVMware class. The package provides access to the vCloud Director SDK through the VCloud class. This SDK is used to retrieve information about virtual machines, virtual applications (groups of VMs) and VM templates. It is also used for the deployment of new virtual machines. By using the vCloud SDK, a user that runs Nebu in combination with VMware is able to monitor his virtual machines through both Nebu and the vCloud web interface.

vSphere

The vsphere package is also used by the DefaultVMware class. This package provides access to the VI Java API. VI Java is an open-source API that communicates with vSphere. This API aims to be more efficient than the official vSphere SDK by providing the same functionality with higher performance and less lines of code.

This API is used to retrieve information about physical resources. This includes the complete physical topology. It is also used to perform VM placement. After a virtual machine has been deployed by the vCloud SDK, VI Java is used to move the machine to the correct physical host. It is also used to change the network storage unit (called Datastore in vSphere) that is used by a virtual machine.

6.2.4. Nebu Application Extension

The application extensions are written in C++ and serve primarily as a proof of concept and as a reference for future extensions. The extensions make up three of the C++ projects, with the C++ version of Nebu-common being the last. The core project for the application extensions is the Nebu-App-Framework. This framework implements common functionality for any application extension, including retrieving topology and virtual machine information. The framework also implements the main application loop, while providing hooks for specific application extensions to perform custom actions.

Figure 6.10 depicts the classes in Nebu-App-Framework and shows the relations between them. The Application and ApplicationHooks classes form the core of the framework. The Application class contains a MainLoop function which periodically calls different components in the system. The ApplicationHooks class contains a list of overridable functions that can be used by specific application extensions to add custom functionality. This design allows for most of the common functionality to be placed in the app-framework while still providing the flexibility to application extensions The execution of the main loop is depicted in Figure 6.12.

Using the app-framework to write a specific application extension requires overriding the ApplicationHooks class and providing specific daemons for the application. An example of this is depicted in Figure 6.11, which describes the extension written for Hadoop.

Figure 6.7: Package diagram of the packages and classes in Nebu-VMware.

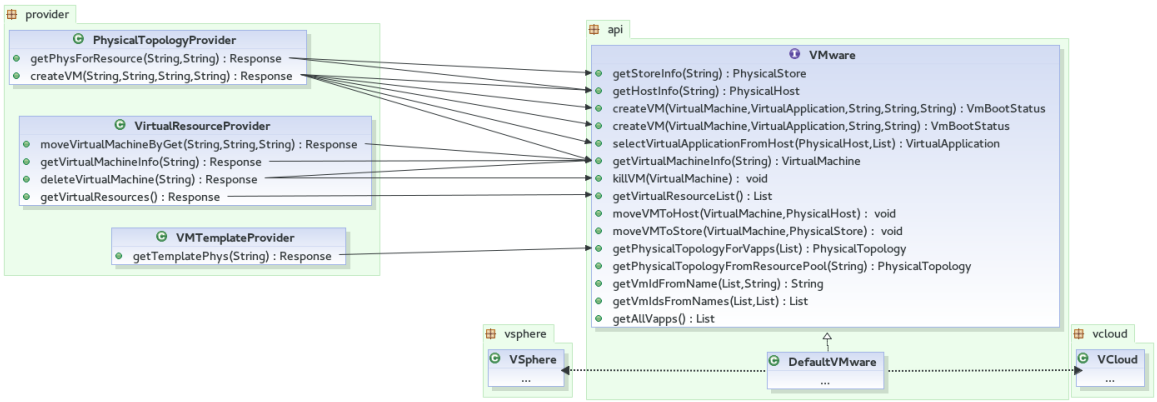


Figure 6.8: Class diagram of the VMware interface and some providers in Nebu-VMware.

6.3. Libraries and Development Tools utilised by Nebu

As in any software project, some of the functionality that the system requires has already been implemented by others and made available in the form of libraries. The external libraries that are used by Nebu can be found in Section 6.3.1. Section 6.3.2 describes the same details for the tools used in the development of Nebu.

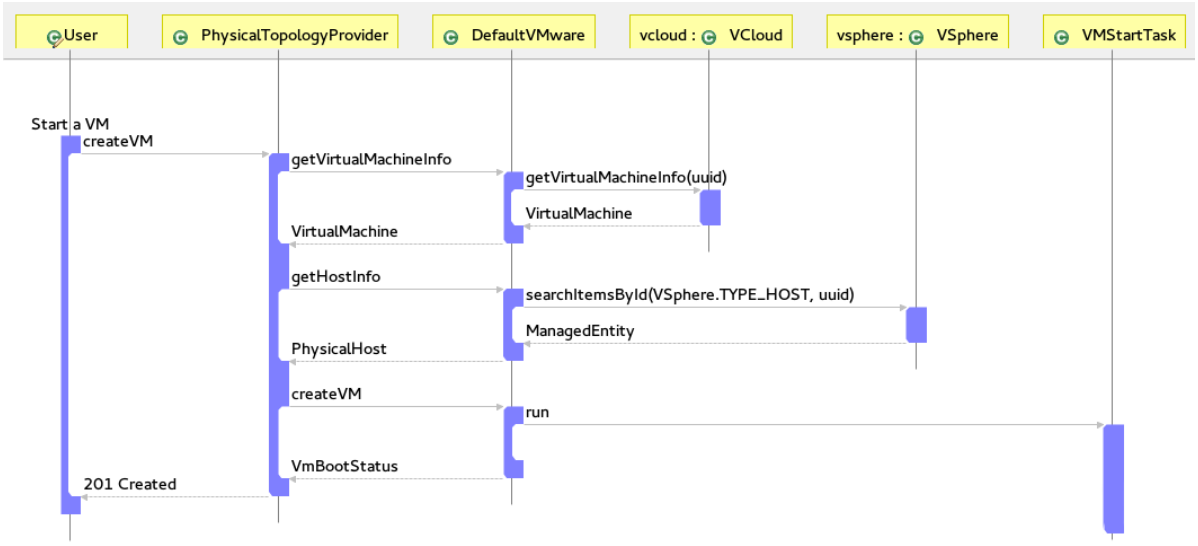


Figure 6.9: Sequence diagram of starting a VM in Nebu-VMware.

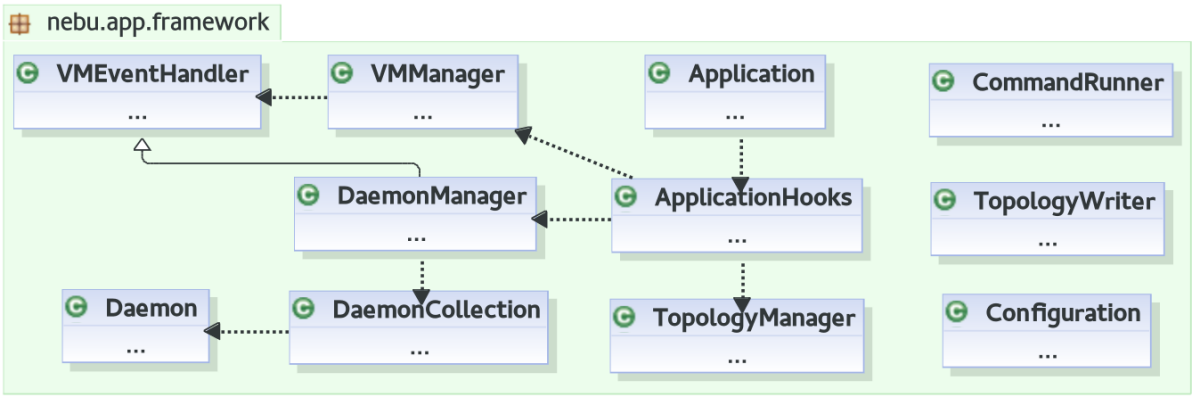


Figure 6.10: Class diagram of the Nebu-App-Framework.

6.3.1. External Libraries used by the Nebu System

Because the complete Nebu system consists of both Java and C++ code, some of the external functionalities required two libraries. One for Java and one for C++. For instance, a library that provides REST-client functionality is required in both the C++ and the Java code. The following list describes all the libraries used in either C++ or Java, why these libraries were chosen, and how this deviates from the original plan (if applicable).

Jersey Library that provides REST server and client functionalities and serves as the reference implementation of JAX-RS (JSR 311 & JSR 339). This library is used because it easy to configure and is used in other projects like Apache ActiveMQ[1] and Apache Camel[2].

Grizzly Jersey needs an HTTP server to provide a RESTful API. Grizzly was chosen to provide this functionality, because it is a simple web server library that is easy to integrate with Jersey.

Guice Although included in the original plan, as it promises a cleaner alternative to the factory-model used for XML conversion, Guice was not used in the implementation. Guice provides dependency injection to allow for clean production code and easy testing, but turned out to be a lot of work

[1] http://activemq.apache.org/
[2] http://camel.apache.org/

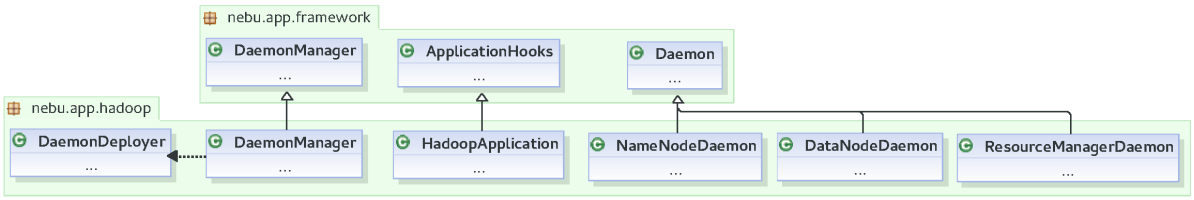


Figure 6.11: Class diagram of the Hadoop extension, with selected classes from Nebu-App-Framework.

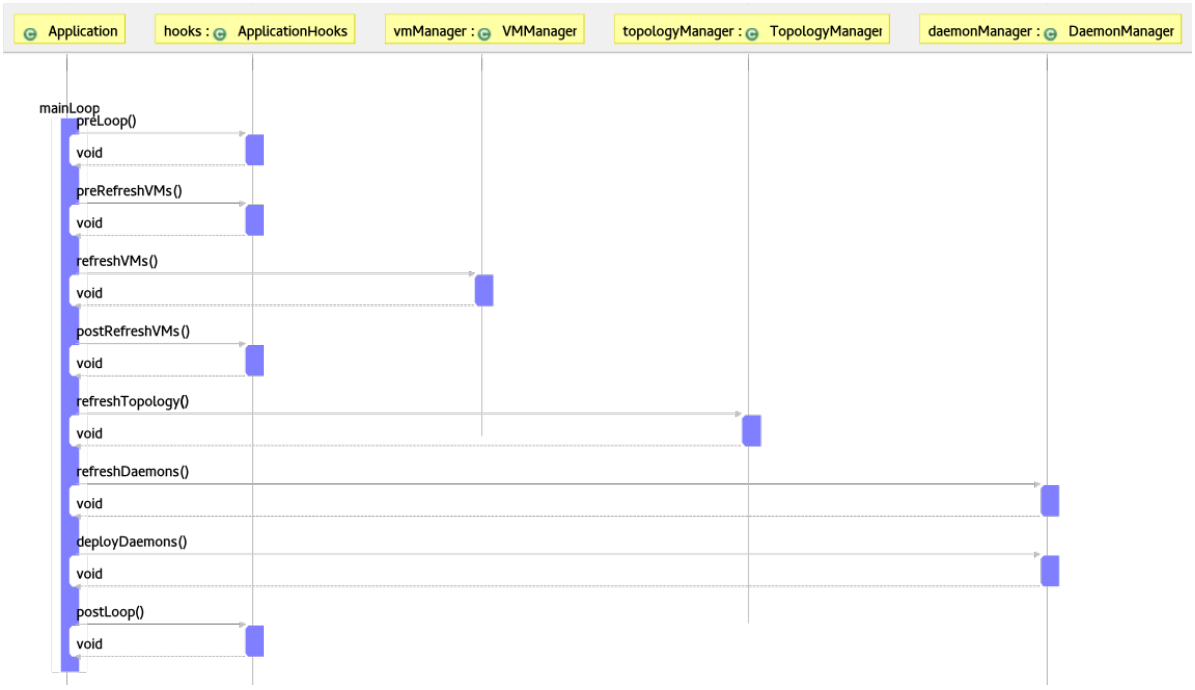


Figure 6.12: Sequence diagram of the main loop in the Nebu application framework. Pre- and post-hooks exist for the `refreshTopology`, `refreshDaemons`, and `deployDaemons`, like depicted for `refreshVMs`. These calls are left out to keep the diagram compact.

to set-up and the benefits were minimal at best. It has therefore been decided to not incorporate Guice into the system.

Log4j An Apache library that provides logging functionality. Not only is Log4j the de facto standard for logging in Java, but it also requires minimum set-up to get a fully working logging system.

vCloud Director SDK To communicate with VMware's API, the official VMware SDK that provides a Java interface to the vCloud REST API is used.

VI Java Library that provides a JAVA interface to the vSphere REST API. This library is used in favour of the official VI SDK[3]. Whilst it provides the same functionality, it requires generally requires less lines of code to achieve the same result. This makes the code more readable and maintainable.

restclient-cpp The C++ applications consume Nebu's RESTful API through the restclient-cpp library. Restclient-cpp is a small open source project, with very few features, but enough to suit the needs of Nebu. An alternative with more support is Microsoft's C++ REST SDK. However, this library uses features of the C++ language that are not available on some enterprise operating systems, as discussed in Section 6.4.1.

[3]https://www.vmware.com/support/developer/vc-sdk/

Language	Purpose	Name & Website
Java	HTTP Server	Grizzly https://grizzly.java.net
Java	Dependency Injection	Guice https://code.google.com/p/google-guice
Java	REST	Jersey https://jersey.java.net
Java	Logging	Log4j http://logging.apache.org/log4j/2.x
C++	MongoDB	mongo-cxx-driver http://docs.mongodb.org/ecosystem/drivers/cpp/
C++	REST	restclient-cpp https://github.com/mrtazz/restclient-cpp
C++	XML	TinyXML2 http://www.grinninglizard.com/tinyxml2/
Java	VMware	vCloud Director SDK https://www.vmware.com/support/pubs/vcd_pubs.html
Java	VMware	vijava http://vijava.sourceforge.net

Table 6.2: External libraries used by Nebu.

mongo-cxx-driver Setting up replica sets and sharding in MongoDB requires connecting to the MongoDB database. Mongo-cxx-driver is the official wrapper for interaction with MongoDB in C++, provided by 10gen.

TinyXML2 The XML used for communication by the Nebu middleware and C++ applications is parsed in C++ using the TinyXML2 library. Through previous experiences this library was known to be easy to use. Other libraries provide similar functionality, so there was no compelling reason to invest time into learning their APIs.

6.3.2. Development Tools used for the Nebu System

The main toolset that was used for the Nebu system contains many common elements, such as an IDE and a documentation tool. This section describes what tools have been used, how they have been used and whether this deviated from the original plan as outlined by the orientation report. The tools are also summarised in Table 6.3.

Eclipse IDE This IDE is a commonly used IDE in Java development and has been used by all team members for the Java code. Unlike the original plan however, a large part of the C++ code has also been written in the Eclipse IDE with the C/C++ plug-in. Eclipse has a complete suite of language features available for C/C++.

Vim editor In addition to the Eclipse IDE, some team members who only made minor changes in the C++ code still stuck with the vim editor. Vim is a popular terminal-based text editor for UNIX-like systems and does not require an extensive setup like an IDE such as Eclipse.

Javadoc and Doxygen Javadoc is the de facto standard for documentation of Java code. Additionally, The Eclipse IDE offers Javadoc support by default. Doxygen was used for the documentation of the C++ code. Doxygen is a popular documentation tool that supports multiple programming languages. Doxygen also support Javadoc-like documentation. This allows documentation throughout Nebu to be consistent in style, independent of the programming language that is used.

Maven Although not mentioned in the original plan, it became apparent that a software project management tool would be required. Two main tools exist for Java projects, Ant and Maven. The team opted to choose the latter since the team has experience with Apache Maven and especially

Purpose	Name & Website
Implementation	Eclipse http://www.eclipse.org/
Implementation	Vim http://www.vim.org/
Documentation	Javadoc http://www.oracle.com/technetwork/java /javase/documentation/javadoc-137458.html
Documentation	Doxygen http://www.doxygen.org/
Building	Apache Maven http://maven.apache.org/
Building	Automake http://www.gnu.org/software/automake/

Table 6.3: Tools used in the development of Nebu.

for Java it is a very popular used option. All Java projects have a Project Object Model (POM) file outlining the required libraries and installation process.

Automake A powerful toolset that allows the buildchain of a project to be configured for a certain platform. This toolset was used for the C++ code, rather than the SCons toolset that MongoDB uses. Whereas SCons offers similar functionality, the make buildchain is more commonly used and better known to the team. Since the projects are independent of the MongoDB code, no restrictions on the choice of buildchain exist.

6.4. Main Challenges

During Nebu development, multiple challenges have been encountered. This section describes the most prominent challenges that were encountered and how these challenges were addressed.

6.4.1. Developing for Enterprise Environments

When developing software, the environment in which the software will run dictates what software and tools are available and under what conditions the software must run. The Nebu project is aimed at enterprise environments, like the environment deployed at Bitbrains, as specified by requirement 7 in Section 2.2.1. For enterprises, server stability and software support are often more important than having the newest technology available. In addition, security is often a high priority, especially when customer data is involved.

One of the most popular operating systems Bitbrains provides to customers is CentOS. CentOS is an enterprise-class Linux distribution with a focus on stability. Core components of the operating system, such as the kernel and system libraries, are generally several years old to and are not patched with additional features throughout the lifetime of the OS. Although this policy improves stability by making support and version control more manageable, it also prevents software written for enterprise environments from using new technology. For example, several C++ libraries that were initially used by Nebu use the C++11 version of the C++ programming language. CentOS is bundled with a version of the C++ runtime that offers no support for the C++11 implementation. This restricts the selection of modern libraries that can be used for software in an enterprise environment.

Enterprise software is also subject to restrictions imposed by security measures, such as firewalls and virtual LANs. Because Nebu performs operations on virtual machines in enterprise environments, security was of importance even during the development and testing phases of Nebu. One of the security measures we took was configuring firewalls in our cluster. Although configuring firewalls is often relatively straightforward, it can be a time consuming process. In particular, Hadoop consists of many services, each with its own set of ports and configurations. Due to one Hadoop service using random ports for communication, without the option to specify a range, firewalls had to be disabled altogether during the Hadoop experiments. For production environments, Hadoop may need to be patched to allow it to function behind a firewall.

6.4.2. Managing Scalable Distributed Systems

Nebu is designed to support automated deployment of distributed applications. Virtual machines can be provisioned using the middleware, after which a Nebu- enabled application, such as Nebu-app-mongo, is responsible for launching and configuring the distributed application. In this process automation and scalability are key.

One of the main challenges in deploying a distributed system on newly deployed virtual machines is managing the process from clean install to an operational application. This includes general system management tasks, such as configuring the operating system and setting up DNS, but more importantly it includes dynamically and programmatically deploying the application in a state that is ready for use. The system management tasks are performed using a combination of *Puppet*[4], and a dynamic DNS solution for enabling hostname resolution. Puppet is a configuration management utility that is commonly used for distributing software and launching services across multiple machines. The deployment of the distributed application is done by Nebu and the required steps differ per application. For Hadoop, Nebu generates configuration files and a file describing the topology. Nebu then uploads these files to the virtual machines and starts the various Hadoop daemons. During runtime Nebu will repeatedly upload new versions of the topology file to reflect changes in virtual machine locations. For MongoDB, Nebu partitions the available virtual machines into replica sets, starts the appropriate daemons, configures the replica sets, and enables sharding for the cluster.

6.4.3. Big Data Application Limitations

Taking advantage of the virtualization awareness provided by Nebu is mostly done in the application layer. Thus, the effectiveness of Nebu depends on support from the application. For Hadoop, Nebu hooks into the Hadoop Virtualization Extensions provided by VMware [6].

MongoDB does not have built-in support for physical topologies, and has several limitations that inhibit dynamically restructuring a distributed MongoDB database. For example, moving a replica from one host to another involves copying all data from one host in the replica set to the new host. In a standard configuration of one MongoDB daemon per (virtual) machine, this could be as much as copying a full disk from one machine to another. This makes dynamic repositioning of replicas slow and impractical. Although it is possible to run multiple daemons on the same machines in different replica sets, this solution requires more complex management and thus is less maintainable. Moving a primary data servers incurs an ever bigger penalty; it triggers a new election in the replica sets, a procedure which can take tens of seconds, during which all data accesses are disabled. This downtime may not be acceptable for all applications, so Nebu does not use runtime reshuffling of data to prevent triggering elections.

Another major limitation in MongoDB is found in its failover mechanisms. When a single node fails, elections can take place in a replica set to chose a new primary server. With two nodes remaining, a majority vote will eventually be reached and the database can resume normal operation. However, if the primary and one secondary server in a replica set fail, it is no longer possible to reach a majority vote as each node must assume there are three participants in the election. The replica set will become unavailable and any query involving that replica set will fail. Bringing the database back online requires bringing the nodes back online, manually reconfiguration of the replica set, and synchronising data between members of the replica set to ensure consistency. In practice this means that having two replicas of a replica set on the same physical host forms a single point of failure that can disrupt the entire database.

Due to the limited time available for the project, there was no time to extend the core of MongoDB with topology awareness or to fix the failover issues. Instead, Nebu works around these restrictions by using topology information during deployment. Nebu guarantees that no two virtual machines on the same host are present in the same replica set.

6.4.4. VMware API Difficulties

The Nebu virtual machine manager (VMM) extension for VMware communicates with the vCloud REST API. This creates compatibility between Nebu and vCloud. Engineers or customers that create their virtual machines using vCloud can use Nebu in the same environment they use for manual VM deployment.

[4] http://puppetlabs.com

Nebu requires knowledge about the physical machines underneath the virtual machines. The vCloud API that is used can provide this information through a so-called 'admin extension'. Unfortunately, this admin extension does not feature fine-grained control on what permissions become available to the user. This means that to use this feature, Nebu needs full administrator permissions on the vCloud instance that is used. During Nebu's development, these permissions were not available.

To solve this, the Nebu VMM extension for VMware requires a second API to be available. It uses the VI Java[5] library to communicate to vSphere and hereby bypasses the vCloud interface. vSphere features a much more advanced permission system than vCloud, and offers a more complete view of the multi-cluster infrastructure. The Nebu VMware extension requires a set of vSphere permissions which are shown in Table 6.1.

6.4.5. Lack of OpenStack Support

Although originally planned, the OpenStack extension for Nebu has not been implemented for this thesis. With the VMware API proving quite difficult to work with and with no easily accessible OpenStack deployment to validate the extension on, there simply was not enough time for the team to implement the OpenStack extension. Setting up an OpenStack deployment in itself is a nontrivial task and with no help readily available from experts, which are available for VMware in the form of Bitbrains engineers, the set-up itself would already have taken at least one full week for one person. Regardless, if the API as prescribed for virtual machine managers is provided by OpenStack, it can be integrated into Nebu without further changes.

[5] http://vijava.sourceforge.net/

7

Quality Assurance

7.1. Overview

Nebu has been designed to be modular and be compatible with other distributed applications and virtual machine managers now and in the future. As is the case for any project of this scale, code quality is important to guarantee maintainability and readability. As Nebu is likely to be further developed by other engineers, and therefore to become a long-lasting software project, measures have been taken to increase code quality. To guarantee this code quality, Nebu's source code is thoroughly tested with the help of unit testing and continuous integration through Jenkins. These measures are described in Section 7.2. The code is also continuously inspected by a code analysis tool called SonarQube. More about SonarQube and other code analytics performed and how they helped to improve Nebu source code can be found in Section 7.3.

7.2. Unit Testing and Continuous Integration

As a project grows, new features are rapidly added, leading not only to an increase in functionality, but also to a decrease in readability and maintainability. A countermeasure that can be taken to help maintainability is the development of a proper test suite containing automated tests. When new features are added, these tests can show whether any previous functionality is broken by the additions. Section 7.2.1 describes the policy that the team applied during the project. Sections 7.2.2 and 7.2.3 detail what testing libraries have been used and how continuous integration was set-up to automatically run the tests.

7.2.1. Testing Policy

The orientation report describes Test-Driven Development (TDD) and how this can be applied to this project. In TTD the policy is to first write a suite of test-cases with stub implementations, only to see the test cases fail. Then, after the full test-suite has been written, the actual implementation of production code is allowed to start. We planned on applying a strategy not unlike this one for this project, with one person writing test code for a certain feature and another independently implementing it. Unfortunately we had to abandon this idea in an early stage of development, as it was unfeasible to apply in this project.

The main issue we encountered is the usage of the VMware API, as well as other frameworks. As it is crucial to mock these APIs to create a clean testing environment, that means that at the time of writing the tests you already need to know what methods are going to be called of these APIs. If the test-code and production-code are written by different team members, both need significant understanding of any external API that is used. This means doing work twice. Using this test driven development methodology would be a very time-consuming operation for this project and as time for actual implementation was already short, the idea was abandoned.

Testing in general was included in the process to a point where most projects now have an extensive test-suite. Especially the Nebu-common library is thoroughly tested, since it forms the basis for the

Language	Purpose	Name	Website
C++	Mocking	googlemock	http://code.google.com/p/googlemock/
C++	Unit Testing	googletest	http://code.google.com/p/googletest/
Java	Unit Testing	Jersey Test-Framework	https://jersey.java.net/
Java	Unit Testing	JUnit	http://junit.org/
Java	Mocking	Mockito	http://code.google.com/p/mockito/
Java	Mocking	PowerMockito	https://code.google.com/p/powermock/

Table 7.1: Overview of Nebu testing libraries.

other Java projects, in that it contains many of the data containers. If new features were to be added to this common library, the tests could clearly show if any of the old functionality has been broken. The only two projects currently lacking in a thorough test-suite are the application extensions. Since these are very small projects that are tightly coupled with the application and rely heavily on a number bash scripts, testing these projects is a complex task that yields few benefits. Simply running the code will already expose any bugs relatively quickly and the amount of time that would have gone into setting up a proper testing environment for these projects was better spent testing the app-framework as this forms the basis for both of the application extensions.

Whereas many different unit tests have been written, no automated system wide integration tests have been written. Although a very time-consuming operation, the scripts that currently exist to manually test the communication between different components could have been expanded upon to feature at least a single test that would test everything from creating an application to launching a distributed application on a virtual machine. The automatic verification through this test would have been problematic however due to restrictions on where the different components can currently run to be able to contact the VMware API.

7.2.2. Testing Libraries

To thoroughly test the Nebu source code, multiple testing libraries have been used. This section describes the libraries that were used and why they were used. An overview of all testing libraries can be found in Table 7.1.

JUnit The de facto standard for unit tests in Java. JUnit allows programmers to specify test cases that validate the results of individual methods.

Mockito A mocking library for Java. Allows users to mock objects to enable testing in isolated testing environments. This way expensive or privileged methods can be mocked (for instance interaction with VMware), to test how the code handles these requests.

PowerMockito A mocking library for Java that is built on top of Mockito. It allows users to mock methods that cannot be mocked by traditional Mockito, such as final or static methods and constructors. Mocking these methods is normally not recommended, but for testing purposes this is sometimes required. Classes or methods might have to be final to prevent extension by third-parties, whilst mocks of these classes or methods are required for testing purposes. This library is used if no other ways of testing a class were available.

Jersey Test-Framework A library that allows unit-testing for Jersey REST client- and server applications. This library is used because Nebu uses Jersey to implement REST APIs.

googletest A testing framework for C++ written by Google. Due to operating system restrictions (see Section 6.4.1), Nebu can not use the latest version of the library, 1.7. Instead, version 1.6 is used and bundled with Nebu.

googlemock Googlemock is a C++ mocking library written by Google to integrate with the googletest framework. It provides both mocking and extended assertions for writing tests. Nebu test code uses googlemock extensively to write unit tests. Like googletest, Nebu is bundled with version 1.6 of googlemock to support CentOS.

Metric	Nebu-common	Nebu-Core	Nebu-VMware	Total Java
Lines	4122	6096	4373	15591
Lines of Code	1931	3335	2094	7360
Comment %	32	26	34	28
Duplicated Lines %	0	0	0	0
Number of tests	311	237	198	746
Line coverage %	87	73	86	80
Issues	1	14	9	24

Table 7.2: The metrics for the Java projects as reported by SonarQube.

7.2.3. Continuous Integration: Jenkins

The creation of test code alone is not enough to have simple checks of whether or not new functionality breaks older code. To verify whether this is case, one needs to actually run the test suite as well. For this purpose a continuous integration server deploying Jenkins [1] has been set-up. The git repositories used for Nebu are configured to notify Jenkins every time new code enters the repository, triggering a build in Jenkins. This ensures that every time new functionality appears on the repository, regression testing in the form of the unit tests is automatically run and results can be seen on the server. Installation of the Jenkins server in itself is simple, as this is all well documented. Some of the Jenkins' plug-ins the team required were slightly more complicated to set-up.

To make the testing environment as realistic as possible, Jenkins was deployed on a virtual machine in Bitbrains' enterprise multi-cluster environment. Just like the virtual machines that Nebu is likely to be deployed on, Jenkins is also running on a CentOS 6.5 image. This allowed for easy testing with older versions of the libraries, as these were the newest ones available on Jenkins.

7.3. Code Analysis

Whereas an extensive test suite can help to improve maintainability and help to quickly see if new code breaks old features, static code analysis can help to improve maintainability, but also to more easily expand existing code. Not only can code analysis help to give a clear overview of the current state of the code, it can also help to find issues in the code. For this purpose a SonarQube-installation has been deployed, of which we describe the installation and results in Section 7.3.1. In addition every Bachelor thesis group has been permitted to submit their code for analysis by the Software Improvement Group (SIG), of which we describe the comments and our response in Section 7.3.2.

7.3.1. SonarQube

To keep track of the current state of the code we deployed a tool called SonarQube[2] that can report on commonly-used metrics in code analysis. It does not only show simple code metrics such as Lines of Code (LOC), but also more complex metrics such as average complexity per method, code coverage and duplication across projects.

All software developed for this project has been submitted to regular analysis, the final results of which are summarised in Tables 7.2 and 7.3. Note that the lines of code mentioned in the tables and figures do not include the test code. For an example of an report by SonarQube, please refer to Appendix D. As the analysis shows, some of the projects still contain what SonarQube labels "issues". The issues that remain are relatively minor however, including issues such as "3 is a magic number" and "avoid commented-out lines of code" (reported for pseudocode). Three of the main statistics (LOC, line coverage and issues) are visually represented for easy comparison in Figure 7.1.

From both the tables and the figure it becomes quite apparent that the applications indeed suffer from very low coverage (that is 0%) due to their lack of tests as explained in Section 7.2.1. It also becomes clear that as the size of the project increases, so generally does the number of issues, though Nebu-core still hosts only 11 issues, of which none are labelled as "Blocking" or "Critical" by SonarQube.

As SonarQube has been analysing the the source code from early in the development stage, it is also capable of reporting changes in code over time. Figure 7.2 shows the lines of code and percentage of

[1] http://jenkins-ci.org/
[2] http://www.sonarqube.org/

Metric	Nebu-common-cpp	-app-framework	-mongo	-hadoop	Total C++
Lines	2238	1034	434	432	4138
Lines of Code	1246	598	257	251	2362
Comment %	15	18	2.3	1.2	13
Duplicated Lines %	0	0	0	0	0
Number of tests	155	57	0	0	212
Line coverage %	79	73	0	0	60
Issues	0	0	0	0	0

Table 7.3: The final metrics for the C++ projects as reported by SonarQube.

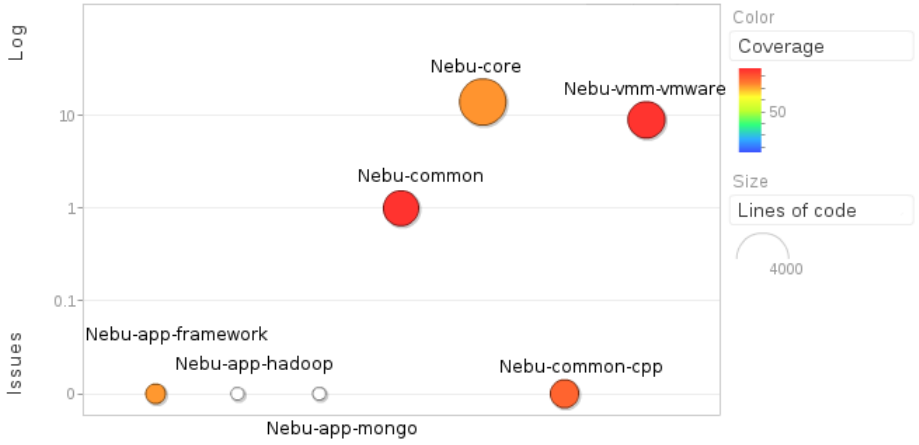


Figure 7.1: Comparison of three main statistics for all projects. The projects are alphabetically ordered on the horizontal axis. The vertical axis represents the number of issues (note the log scale). The size of each circle indicates the lines of code in each project and the colour indicates the coverage, where fully red equals 100%.

comments over time for the Nebu-common library implemented in Java. As can be seen, the percentage of comments has been relatively constant throughout the project, indicating that new source code was already well commented when committed to the server.

Another metric worth exploring is the ratio of testcode compared to production code, but unfortunately SonarQube has no tools to provide this metric. When excluding commented lines, a rough estimate made using a word count utility yields a ratio of production-code:test-code of 1:1.2. SIG recommends a ratio of at least 1:1, which is less than the estimate value of the combined Nebu projects.

7.3.2. Software Improvement Group

The Software Improvement Group (SIG) is a company that translates detailed technical findings concerning software systems into actionable advice for upper management. SIG performs static analysis on the Nebu source code two times. The first time to provide feedback to the development team on what can be improved. The second time to confirm that this feedback is taken in consideration and to evaluate the final quality of the code. Feedback from SIG can be found in Appendix B.

SIG awards point on a ranking wherein five stars is the maximum. The points awarded are based on items such as the presence of testcode, the amount of code duplication, module coupling, and other metrics indicating good readability, maintainability and testability. In the initial feedback, the implementation scored four out of five stars, indicating the code was already up to high standards. The main issues reported by SIG were duplication among the different C++ projects and the module coupling with classes like VirtualMachine carrying too much responsibility and being used too much.

The first issue, that of duplication, has been resolved by the introduction of the App framework, that removes all duplication from the two projects (as the most recent SonarQube measurements indicate in Table 7.3). The module coupling has also been addressed, by moving the Builder classes, which were inner classes of all the objects they build, to separate classes as suggested by SIG. In addition

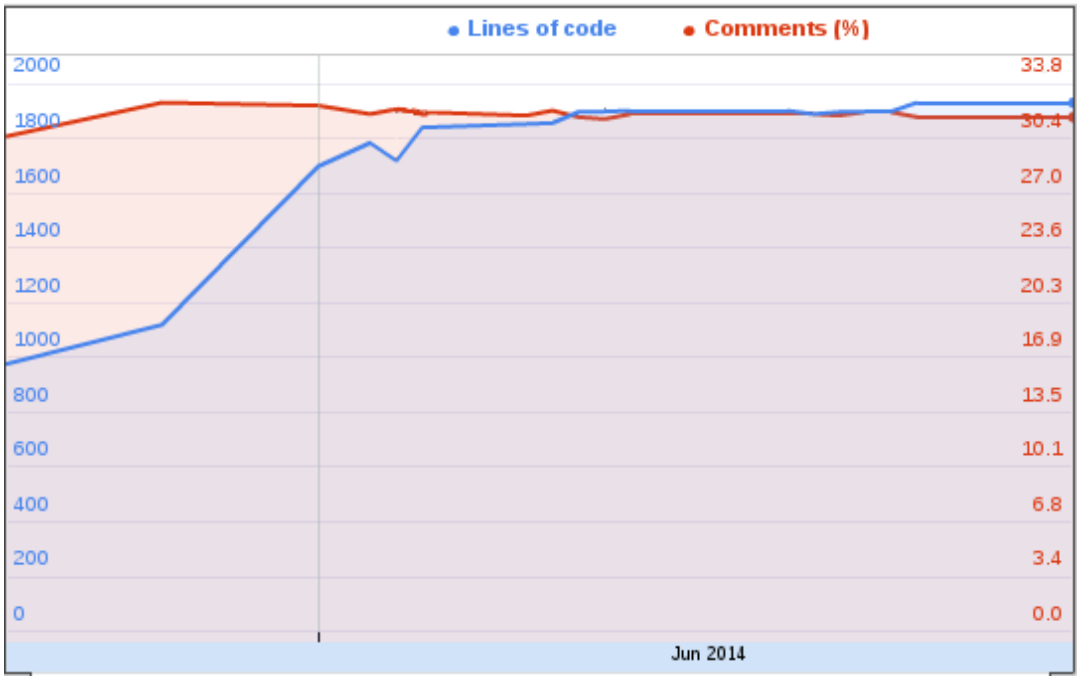


Figure 7.2: Lines of code (blue) and percentage of comments (red) over time for the Java version of the Nebu-common library.

the builders were rewritten to follow a similar inheritance pattern as the containers classes to further reduce duplication. These and other points of refactoring resulted in a very positive second round of feedback from SIG, in which they remark that from their observations it is clear the recommendations of the initial evaluation were addressed properly in the rest of the development.

8

Experimental Work

8.1. Overview

To validate the system, Bitbrains provided access to part of their multi-cluster environment for conducting experiments. Table 8.1 shows information on the virtual machine image, as well as the version of MongoDB and Hadoop that were used in the experiments. For these experiments two racks have been provided, within the same data centre, both of which are in a live production environment. All experiments have been conducted using eighteen virtual machines as data nodes for Hadoop and data servers for MongoDB. In addition a name and resource node in the case of Hadoop and three config servers and a queryrouter for MongoDB have been used.

The main focus of the experiments is to validate Nebu as a solution to the issue of reliability of distributed applications in a virtualized environment, as set out by the first research question. To this end, the reliability and performance of both Hadoop and MongoDB are tested using the various policies provided by Nebu.

For Hadoop, a subset of the HiBench benchmark suite [7] has been used to test both the reliability and performance. Only the TeraSort and WordCount benchmarks worked with the latest version of Hadoop, so these were both run. For reliability purposes the choice of workload is not highly relevant, as the Hadoop replication policy does not distinguish between different types of data blocks. Result data in any form or shape will therefore be treated in the same manner, which means small differences are to be expected between the different workloads as far as duplication spreads are concerned. For performance analysis there are few workloads available from real world scenarios, it is therefore common for Big Data research to focus on synthetic workloads, such as WordCount and TeraSort described here. However, previous work on the BTWorld use case [5] is used to validate that the reliability of the system does not degrade when taxed with a complex, real-world workload. The key parameters of each benchmark is summarised in Table 8.2.

The sizes of the workloads used for WordCount and TeraSort are of such magnitude that two important criteria can be met. Firstly, as each virtual machine has only 4 GB of memory and each node will store 10 GB, it is impossible for the machine to fully fit the data into memory. This ensures that data locality and in memory transfers can actually affect the efficiency of operations. Secondly, as the data is split into blocks of 128 MB and each block is replicated trice, each node will hold on average 240 blocks of data. This is a sufficiently large number to see differences in the number of physical hosts that share the same data. Whereas larger data sets might have offered even larger and perhaps clearer differences in these numbers, the fact that the experiments run in a production environment limits the amount of resources (both physical resources and time) available for experimentation. In this set-up experiments already had to be carefully planned and communicated so that any alarms raised at Bitbrains due to heavy loads on the system would not be a cause for concern.

To analyse the influence of topology awareness, three levels of awareness are tested; no awareness (NA), rack awareness (RA), and VM awareness (VA). In addition, three placement policies are tested to identify how they influence the reliability and performance of the system.

The performance of MongoDB is tested using the Yahoo! Cloud Serving Benchmark (YCSB) [8]. The YCSB consists of multiple common workloads in cloud applications, and is used for benchmarking

Property	Value
Operating System	CentOS 6.5
Memory size	4 GB
Storage capacity	128 GB
Hadoop	2.4.0
MongoDB	2.6.3

Table 8.1: Properties of the Virtual Machine image and the versions of the Hadoop and MongoDB.

Benchmark	Size	Mappers	Reducers
WordCount	180 GB	720	360
TeraSort	200 GB	800	400
BTWorld	100 GB	Varying	Varying

Table 8.2: Parameters for the workloads used to test Hadoop.

database systems. The goal of these experiments is to identify performance penalties when adding virtualization awareness to the deployment of MongoDB. Due to the time-consuming task of manually setting up a sharded replicated MongoDB cluster, only one policy was tested. The setup tested is a deployment using the Locality policy (six hosts, three virtual machines per host). For the experiment without virtualization awareness, three virtual machines on a single host were combined to form a replica set. For the virtualization aware experiment, Nebu spread the replicas over different hosts.

For all of the experiments the Ganglia Monitoring System[1] was also set-up to monitor metrics, such as CPU load, memory usage, network transfer, and I/O transfer. Ganglia specialises in monitoring distributed systems, with nodes posting information to a central server that can aggregate and display the information. Unfortunately the graphs obtained from Ganglia offer few new insights other than being evidence to the high network speeds Bitbrains can support between different hosts. The resulting graphs have therefore been excluded from this report.

8.2. Results

In this section, the results of the various experiments are discussed. The reliability and performance tests for Hadoop are described in Section 8.2.1 and Section 8.2.2, respectively. For MongoDB, the reliability and performance are described in Sections 8.2.3 and 8.2.4.

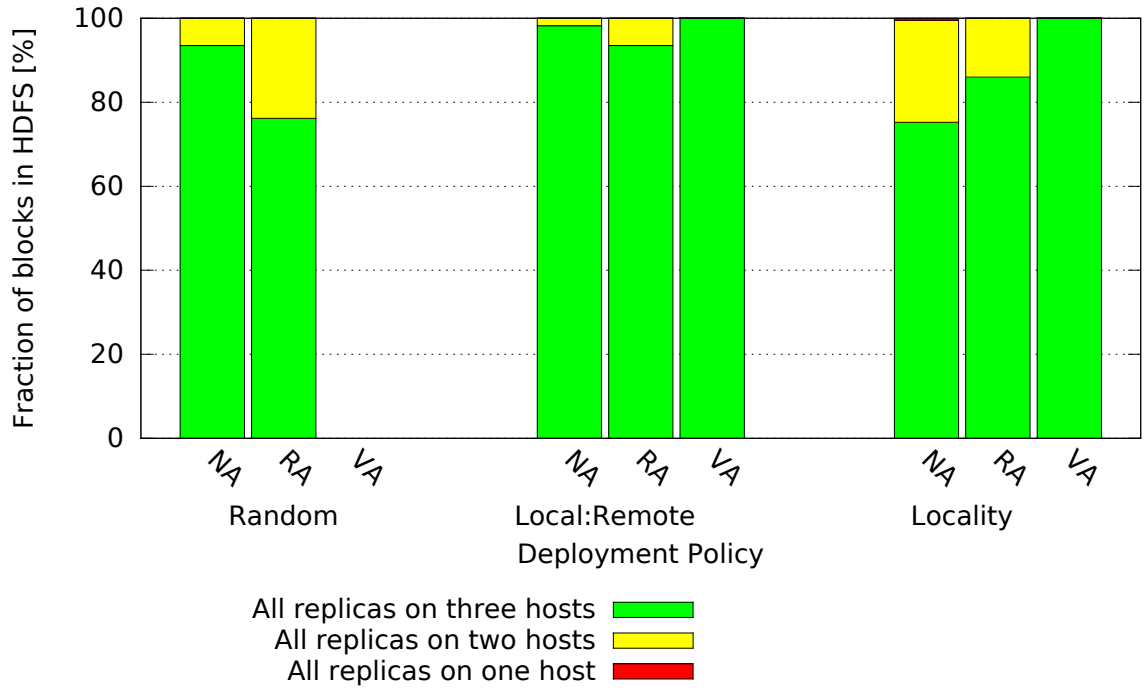
8.2.1. Reliability Hadoop

To demonstrate the effects of topology-awareness on the reliability of Hadoop, the locations of replicas have been monitored throughout the execution of the HiBench benchmark. The `fsck` command provided by HDFS was used in three-minute intervals to record the exact placement of each block and each replica. Analysis of the replica placement revealed that no replicas were moved throughout the experiments after initial placement. To provide comprehensive results, only the snapshots at the end of both the WordCount and TeraSort benchmarks are presented in this section. These snapshots describe the largest amount of blocks and are thus statistically the most meaningful.

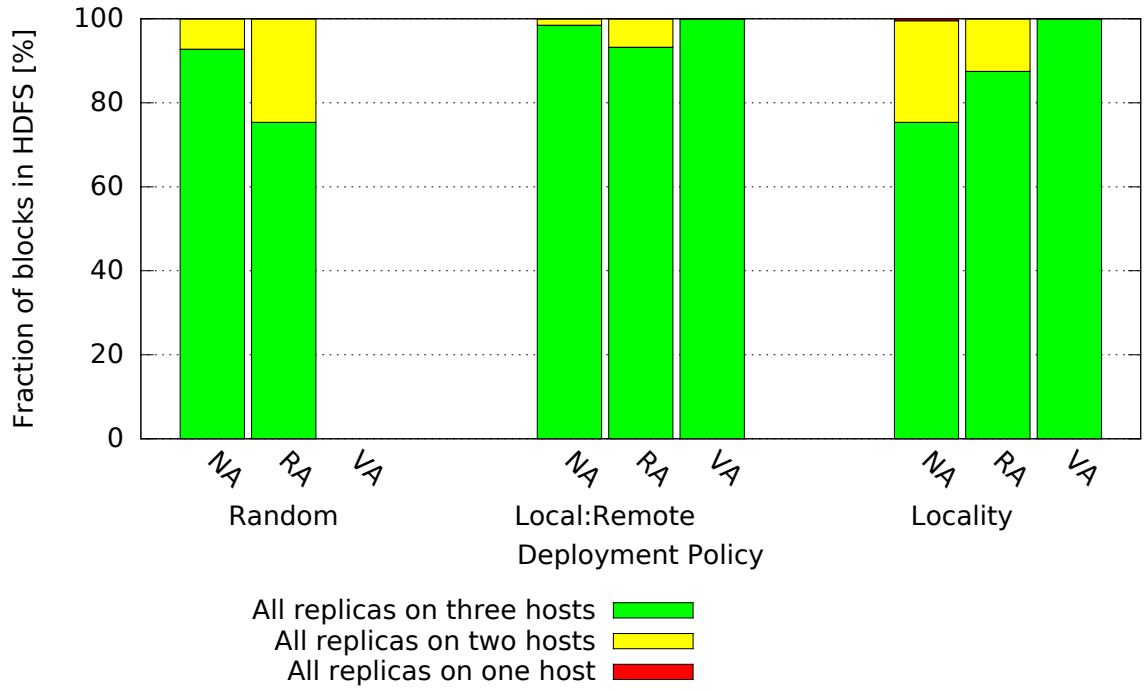
The results of the replica placement analysis are depicted in Figure 8.1. Three aspects deserve explicit mentioning. First, the differences between the WordCount and TeraSort results is small. For example, the figures show that for the Locality policy without topology awareness roughly 0.7% and 1.0% of blocks are placed on a single host after the WordCount and TeraSort experiments, respectively. Second, the results show that the rack-aware HDFS in general places two replicas on the same host more often than HDFS without any topology awareness. For the experiments with the Locality policy, the rack-aware HDFS places two replicas on the same host less frequently. Finally, during the virtualization aware experiments not a single block had two replicas placed on the same host.

Similar results were seen throughout the execution of the BTWorld workload (not depicted). Due to time and resource constraints, this benchmark was only run once using an altered Locality policy to spread the load over the two available racks. Although six hosts were used with three virtual machines each, as per the default Locality policy, these six hosts were spread over two racks. This change does

[1] http://ganglia.sourceforge.net/



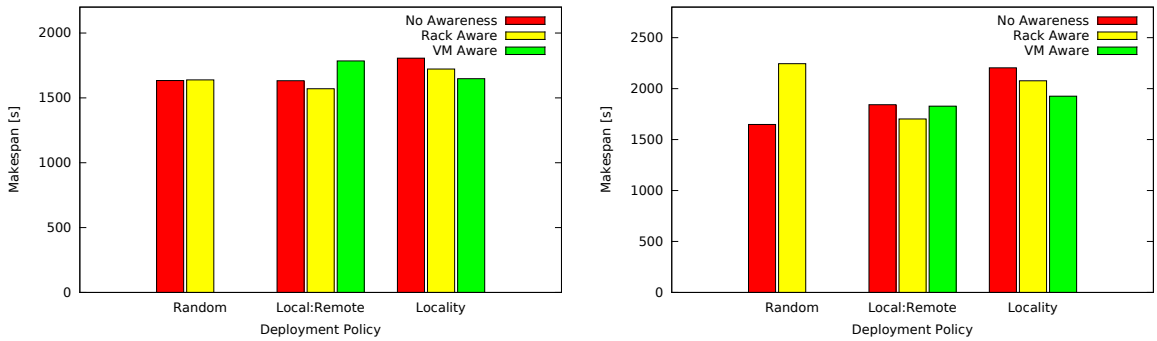
(a) Replica distribution for WordCount experiments.



(b) Replica distribution for TeraSort experiments.

Figure 8.1: Replica distribution on HDFS after each experiment. Each bar corresponds to a single placement policy with a specified level of topology awareness and depicts what fraction of blocks have their three replicas stored on one, two or three physical hosts. The topology awareness levels are: no awareness (NA), rack awareness (RA), and virtualization awareness (VA).

not impact the effect of single host failure. Like the HiBench experiments with virtualization awareness, no blocks were detected with two replicas on the same physical host throughout the BTWorld workload.



(a) Makespan for WordCount on 180 GB input.

(b) Makespan for TeraSort on 200 GB input.

Figure 8.2: Makespan for the execution of the WordCount and TeraSort benchmarks on 18 nodes.

8.2.2. Performance Hadoop

Figure 8.2 depicts the makespan for the WordCount and TeraSort experiments. In general, the results do not vary significantly between topology awareness levels. The single exception to this trend is the rack-aware TeraSort using the Random placement policy. The rack-aware run took about 36% longer to complete. For the Locality policy, providing virtualization awareness decrease the makespan of WordCount and TeraSort by 8.9% and 13%, respectively, when compared to the baseline without any virtualization awareness. Rack awareness also caused a decrease in makespan when using the Locality policy, despite all nodes being contained in a single rack for this experiment. Finally, the only experiment where virtualization awareness caused a decrease in performance is the WordCount running on Hadoop deployed by the Local:Remote policy. In this case a 9% increase in makespan was observed.

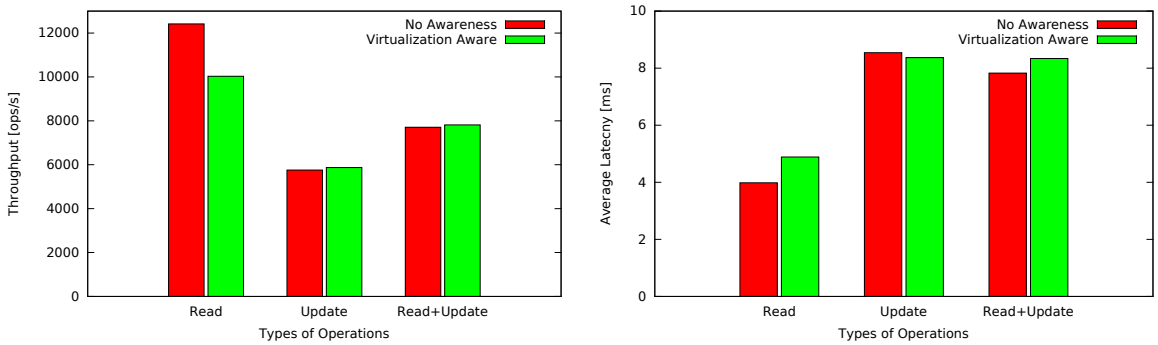
The performance of the BTWorld use case was recorded as well. However, no baseline could be set due to the intensity and duration of the workload, so no comparison can be made.

8.2.3. Reliability MongoDB

As the placement of replicas in a MongoDB cluster is determined at the time of deployment, running benchmarks will not reveal relevant information regarding the reliability of MongoDB in a virtualized environment. To gain insight into the effects of node failure on a MongoDB cluster, failure injection was used on a live deployment of MongoDB. After inserting records into a sharded replicated cluster, numerous combinations of MongoDB daemons were killed. This process led to the following conclusions on the reliability of MongoDB, associated with four types of failure:

1. Removing secondary nodes does not impact the availability of the cluster.
2. Removing a primary node leads to unavailability of data until a secondary node of the same replica set is promoted to primary. This is an automated process that usually completes within a minute.
3. Removing a majority of the nodes in a replica set, including the primary node, leads to unavailability of data and requires manual restoration of the cluster. Any query attempting to access the lost data is stalled or fails to complete.
4. Removing all nodes in a replica set leads to unavailability and loss of data, unless at least one of the lost nodes can be restored.

Failures of the first two types in the list do not significantly hinder the operation of a MongoDB cluster, and can occur regardless of virtualization awareness. The third and fourth type of failure, however, require human attention to keep the data accessible. The likelihood of these failures occurring given a single physical host failure can be estimated through simulation. For a deployment using the Locality policy (six hosts, three VMs per host), these simulations reveal that for a single host failure the chance of a type 4 failure is 0.74% (std. dev.: 0.012), while the chance of a type 3 failure is 22% (std. dev.: 0.050). Although the simulation is a simple loop of a random generated placement of replicas, the used code is available upon request. During experiments, Nebu did not place multiple replicas of the same set on a single host, so only type 1 and 2 failures could have occurred.



(a) Throughput in operations per second for various artificial workloads.

(b) Average latency per operation for various artificial workloads.

Figure 8.3: Results of running a subset the YSCB benchmark on a MongoDB cluster deployed using the Locality policy with 18 data servers. All workloads were run three times with one million operations per run. Averages are reported.

8.2.4. Performance MongoDB

Figure 8.3a depicts the throughput for a read-only, an update-only and a combined workload. For the read-only workload the throughput of the virtualization-aware deployment is 19% lower than the deployment without awareness. For the update-only and combined workloads, virtualization awareness improves the performance slightly, respectively with 2.0% and 1.4%. Figure 8.3b depicts the average latency of operations during execution of the workloads. The difference is again largest for read operations, with an increase of 23% when adding virtualization awareness. Update latency decreases by 2.0%, while the combined workload reveals an increase of 6.5% when adding virtualization awareness.

8.3. Discussion

The experiments for Hadoop validate the claim that Nebu can improve the reliability of Hadoop without degrading performance. Throughout all virtualization-aware experiments there was not a single block that had multiple of its replicas stored on the same physical host, whereas this did occur for all experiments with rack-awareness or without any topology awareness at all. This can be intuitively explained by the fact that there were at least three physical hosts used for each experiment, which means that there was always a viable placement of replicas without them residing on the same host. The generally poorer placement of a rack-aware Hadoop can also be explained intuitively. When placing replicas without any topology awareness, Hadoop can pick three random nodes to place its data on. Rack-awareness adds the restriction that the second and third replica must be placed in a different rack than the first replica, and thus these two replicas are placed in the same rack when only two racks are available. This limits the possible options for placement, increasing the chance of having two replicas end up on the same host.

The performance experiments are not as conclusive as the reliability experiments, but do not point towards a loss in performance when adding virtualization awareness. The makespan differences when adding virtualization awareness range from an increase of 9% to a decrease of 13%. The experiment closest to VMware's experiments in the Hadoop Virtualization Extensions white paper [6] is the experiment using the Locality policy. Although this experiment did not span two racks, the colocation of multiple virtual machines on the same host allows for increased data locality. The makespan decrease of 9% and 13% for WordCount and TeraSort, respectively, rival the 13% decrease in makespan observed by VMware.

The evaluation for MongoDB is less valuable, as no changes have been made to the operation of a MongoDB cluster. Nebu adds the ability to deploy MongoDB clusters automatically and it does so using information about the physical topology. Simulations of the reliability of MongoDB reveal that nearly one in four single host failures lead to the inaccessibility of a manually configured MongoDB cluster, when the virtual machines are placed according to the Locality policy. Using the same policy, Nebu is able to prevent this scenario altogether by ensuring that multiple replicas of the same replica set are spread over multiple hosts. The performance of MongoDB decreased for read operations when switching from a manual to an automated, virtualization-aware deployment. However, the chosen

manual configuration is assumed to be the ideal case for performance, as the primary servers are spread evenly over different physical hosts. It is likely that different manual or randomised placements of replicas achieve different performance. Evaluation of the performance using different placement is left as future work.

Overall, the experiments reveal that Nebu greatly improves reliability of both Hadoop and MongoDB by preventing a single host failure from impacting the availability of the system. The performance evaluation is incomplete due to the difficulties in evaluating distributed systems. However, the observed performance loss is insignificant when considering the benefits of keeping data safe. Customers at Bitbrains, and enterprises in general, tend to prioritise reliability over performance, as failure can be catastrophic for business operations.

These experiments were run in a production environment. Despite the relatively low load from other customers at the time of running the experiments, it may have influenced the results, so more experiments need to be done before definitive claims about the performance can be made. Under higher load, or with a larger portion of hardware available for experiments, shared I/O like network and hard disk accesses may become a more apparent bottleneck.

9

Ongoing and Future Work

9.1. Overview

This section is split into three major parts. First, Section 9.2 describes new functionalities that should be implemented in future versions of Nebu. Second, Section 9.3 describes how the Nebu source code will be open sourced, and a community can be formed to further develop Nebu. Third, process improvements are described for future projects in Section 9.4.

9.2. New Functionality

The Nebu project is designed to grow over time and support many applications and virtual machine managers. This section describes the features that need to be implemented to turn Nebu into a production-ready application. This excludes extensions for applications or virtual machine managers that are not yet supported. Nebu does not rely on these extensions to be production ready.

9.2.1. Authorisation System

As described in Section 4.4.2, the authorisation system is a requirement that was introduced during development, rather than the orientation phase. Because Nebu can serve multiple users at the same time, it is important that each user can only access their own information. Currently, Nebu does not have an authorisation system, which means that every user can potentially obtain information about the virtual machines of other users. To be called production ready, Nebu must feature an authorisation system. However the structure of the API and implementation makes the introduction of authorisation-based access simple.

As all API requests focus around applications in the Nebu system and all requests internally go through the AppsProvider class (see Section 6.2.2), this is the only place where authorisation verification would be required. Other than this a login URI would have to be provided, as well as permanent storage of user credentials.

9.2.2. Fault Tolerance

Nebu provides information to other distributed applications which allow them to improve their reliability and fault-tolerance. However, Nebu itself currently has no built-in fault-tolerance mechanism.

To provide Nebu with fault tolerance, two main features need to be added. First, a database should be introduced to provide Nebu with persistent storage. This prevents data loss in case of node failure. Second, a communication system should be introduced between multiple Nebu instances. Using multiple Nebu instances at the same time increases the availability of the system in case of node failure.

Fault tolerance is not required for Nebu to become production ready. Nebu is currently only used for the initial deployment and placement of distributed application. It does not perform actions during the runtime of these applications. For this reason, persistent data about these applications is not required.

9.2.3. Hadoop Network-Storage Awareness

Nebu's Hadoop extension provides information about the physical topology to Hadoop. Hadoop can automatically improve performance when supplied with physical topology information. However, Hadoop assumes that the virtual machines use local disks, which might not be the case in a commercial cloud. To further improve Hadoop's performance in a virtual environment, network-storage awareness must be built into Hadoop.

Although this feature is part the Hadoop extension, and not of the Nebu-core, the addition of this feature is important to make Nebu production ready, because this extension provides proof of Nebu's capabilities and contributions.

9.3. Publication of the Nebu Code

Although the work the team has done on Nebu for the thesis is now finished and a working prototype has been delivered, the possibility for Nebu to gain new functionality remains. It has been agreed with Bitbrains that the software written for Nebu will be open sourced through the popular open source code portal GitHub. The working prototype of the Nebu system, including all of the different projects, will be released there with accompanying documentation and a reference to this thesis.

The documentation will describe how Nebu can be installed and deployed in a virtualized environment, including a description of where the different components should be placed. In addition, short 'ReadMe' files will be provided for the individual projects, outlining how a new placement policy can be added to the Nebu-core framework, or how the App-framework can be used to easily integrate other distributed applications into Nebu. The RAML specifications for both of the APIs will also be publicised, so that developers for other virtual machine managers have a clear overview of what the API needs to offer to be integrated with Nebu.

With the code publicised other developers can provide patches and new functionality to Nebu, which will go through a pull request mechanism supported by GitHub. Developers will submit their new code for examination by a contributor (in this case either a member of the team, or possible a Bitbrains engineer if they continue Nebu's development), after which the code can either be merged into the main branch, or be rejected specifying why it is not good enough to be merged into the project. This way quality can be maintained, whilst also providing sufficient opportunity for other developers to contribute to the project.

9.4. Process Improvements

In general, the processes described in Chapter 4 proved to be effective. There are not a many things that the team would like to change. For future projects, however, we will make sure to start the experiments at an earlier stage. In this project, the experiments started in the 8th week of the 10-week project. However, the first week of experiments got lost due to system permission problems that obstructed our software from working properly in the enterprise environment. Additionally, multiple experiments failed and needed to be restarted. This consumed a significant amount of time.

The team overcame these issues during the final weeks of the project. The planning was constructed in such a way that other tasks could be rescheduled and time was made available to properly perform the experiments.

10

Conclusion

Petabytes of data are processed daily by distributed applications built upon Hadoop and MongoDB. A significant fraction of these applications run in virtual environments using cloud infrastructure to cope with this vast amount of data. Many commercial clouds use virtualized environments. When distributed applications are run in virtualized environments, the reliability and fault tolerance of data storage by the application can no longer be guaranteed. Traditionally, fault tolerance mechanisms have been based on the assumption that the failure of a single node in a system is not correlated with the failure of others. In virtualized environments, this assumption no longer holds.

The research questions posed at the start of this thesis, can now be answered as follows:

1. To improve reliability of virtualized applications, we design and develop Nebu, a topology-aware deployment system for reliable virtualized multi-cluster environments. Nebu fetches topology information from virtual machine managers, and provides this information to distributed applications. Two distributed applications, MongoDB and Hadoop, have been extended to use this topology information in their placement strategies, to guarantee reliability without loss of performance. In addition, Nebu allows automated deployment of a complete cluster of virtual machines for a distributed application in only four API calls, whereas this deployment is currently done manually, which is a labour intensive tasks. This deployment is done through user-specified policies, to attain the goal of performance and/or reliability.

2. To validate the design of the final product, experiments have been conducted in a real-world production environment. These experiments show that the Nebu system can help applications provide reliability whilst seemingly not degrading performance. The introduction of virtualisation awareness has been shown to ensure that data duplication uses the maximum number of hosts for its placement.

The main contributions outlined of the Nebu system and the work done for this thesis can be summarised as follows:

1. A user study through interviews of Bitbrains employees, as reported in the orientation report.

2. Nebu, a generic software ecosystem to run distributed applications with virtualization awareness as described in Chapters 5 and 6. An overview of how Nebu fulfils the requirements from Section 2.2 is given in Table 10.1.

3. An efficient API for engineers and end-users to create a virtual cluster and deploy a distributed application, as described in Chapter 5.

4. An evaluation of the impact Nebu has on both the reliability and performance of multiple big data platforms. Both experimental and statistical evaluation is described in Chapter 8.

#	Critical	Requirement met	Description
1	YES	YES	Nebu provides a generic physical topology model in Section 5.5.
2	YES	YES	Nebu provides compatibility for new distributed applications through its RESTful API. See Section 5.4.1.
3	YES	YES	Nebu provides compatibility for new virtual machine managers through its RESTful API. See Section 5.4.2.
4	YES	YES	Nebu core and Nebu VMM extensions provide this feature as part of their respective APIs. See Section 5.4.1 and Section 5.4.2.
5	YES	YES	Nebu allows users to select from multiple placement policies and configure deployment specifications by hand. See Section 5.6.
6	YES	YES	Nebu core and Nebu VMM extensions provide this feature as part of their respective APIs. See Section 5.4.1 and Section 5.4.2.
7	YES	YES	Nebu is developed in an enterprise environment and solely uses external libraries that also run in this environment. See Section 6.4 and Section 6.3.1.
8	NO	NO	Nebu does not feature an authorisation system. See Section 9.2.1.
9	NO	YES	Nebu provides extensions for MongoDB and Hadoop. See Section 6.2.4.
10	NO	PARTIALLY	Nebu provides an extension for VMware. See Section 6.2.3.

Table 10.1: Nebu system requirements and how they are satisfied.

Bibliography

[1] D. Borthakur, *The hadoop distributed file system: Architecture and design,* Hadoop Project Website **11**, 21 (2007).

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, *Apache hadoop yarn: Yet another resource negotiator,* in *Proceedings of the 4th annual Symposium on Cloud Computing* (ACM, 2013) p. 5.

[3] J. Dean and S. Ghemawat, *Mapreduce: simplified data processing on large clusters,* Communications of the ACM **51**, 107 (2008).

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, *Mesos: A platform for fine-grained resource sharing in the data center,* in *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (2011) pp. 22–22.

[5] T. Hegeman, B. Ghit, M. Capota, J. Hidders, D. Epema, and A. Iosup, *The btworld use case for big data analytics: Description, mapreduce logical workflow, and empirical evaluation,* in *Big Data, 2013 IEEE International Conference on* (IEEE, 2013) pp. 622–630.

[6] VMWare, inc, *Hadoop Virtualization Extensions on VMware vSphere 5,* http://www.vmware.com (2014).

[7] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, *The hibench benchmark suite: Characterization of the mapreduce-based data analysis,* in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (IEEE, 2010) pp. 41–51.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking cloud serving systems with ycsb,* in *Proceedings of the 1st ACM symposium on Cloud computing* (ACM, 2010) pp. 143–154.

A

Orientation Report

Orientation Report

Jesse Donkervliet, Tim Hegeman, Stefan Hugtenburg

July 21, 2014

Summary

Bitbrains is a Dutch IT company that provides cloud solutions to a variety of customers, such as the Dutch bank ING. Many of these customers run distributed applications such as the processing framework Hadoop, or the database MongoDB on top of the virtualized environment presented by VMware. Since these applications are not virtualization aware, data replicas can still be placed on the same host. This can lead to a degradation of both performance and reliability. Our project involves creating software that offers virtualization awareness to distributed applications. VMware has already created an extension for HDFS that allows it to use information on the physical topology. However, this solution only works for the HDFS/VMware combination and not for other applications such as MongoDB.

We have conducted interviews with multiple people at Bitbrains, which have resulted in several requirements to the system. These requirements can be described as placement suggestions to the VM manager before deployment, influence the scheduler of the VM manager at runtime, and the forwarding of topology information to the distributed applications. To fulfil these requirements we will create a middleware platform, that communicates with both the distributed application and the VM manager. The APIs used between these extensions and the middleware will be RESTful, which is a web-based standard for APIs. To keep our code portable, Java will be used as the main programming language. To keep our code maintainable, we will use test-driven development and continuous integration.

1 Introduction

Bitbrains is a company that specialises in providing cloud solutions to a variety of customers, among which are the Dutch bank ING and the Dutch insurance company Nationale Nederlanden. Many of these customers use distributed computing frameworks, such as Hadoop, or other forms of distributed applications, such as MongoDB. These applications run in virtualized environments through the use of VMware. Bitbrains is interested in increasing virtualization awareness in these distributed applications, so that caveats such as replication on the same physical host can be avoided. This report summarises the orientation phase of this project, during which we familiarise ourselves with the field and investigate what libraries, APIs, etc. are best suited for our product.

We will provide a product that functions as a middleware layer that allows distributed applications to become virtualization aware. This virtualization awareness provides applications with data-placement and scheduling policies that aim to realise a specified level of fault-tolerance and performance in a virtualized environment. To demonstrate the generality of our middleware we will provide extensions for two distributed applications and two virtual machine managers. The first application that we incorporate in our product is MongoDB [1], a distributed document-based database. On the other side of our middleware we will connect to VMware [2], a platform that provides virtualization services. Both of these systems are used by Bitbrains and the rest of the industry and both systems will require a small wrapper/extension that allows for interaction with our middleware. The second pair of applications that will be added are HDFS, Hadoop's [3] Distributed File System, and OpenStack [4], an open-source virtualization platform.

In this orientation report we summarise the results of the orientation phase of this project as follows: first in Section 2 we give some more detailed information on the applications that we will extend and on similar products in this field. Section 3 discusses some related work and existing solutions for the problem

described earlier. Section 4 focuses on the requirements gathering for this project, including a description of the various interviews we have conducted as well as the main requirements we identify. Section 5 then describes our chosen approach based on these requirements, including what language and tools we will use. Finally Section 6 details what quality guarantees we will provide and how these are verified.

2 Background

In this section we will further describe the applications that our product will incorporate, two at each side of the middleware. On one side we find two distributed applications: MongoDB and HDFS, which are further detailed in Sections 2.1 and 2.2 respectively. On the other side of the middleware we find two virtualization platforms: VMware and OpenStack which are described in respectively Section 2.3 and Section 2.4.

2.1 MongoDB

MongoDB is a distributed document database, using a NoSQL-structure for information retrieval and alteration. In contrast to the table-based model of SQL-databases, MongoDB utilises a dynamic JSON-like document store they call BSON. MongoDB has several features to support its use as a distributed application, including replication and sharding. Replication is used to ensure high availability of data, by placing multiple copies of a single data set on multiple machines. Additionally, replication can be used to spread data geographically and thus reduce latencies to applications deployed worldwide [5]. Sharding is used by MongoDB for horizontal scaling. A single data set is split into shards and each shard is placed on a different machine. Queries on the data set are redirected to the relevant shards through a query router and results are then aggregated.

A typical distributed MongoDB database consists of three types of services: a set of shards (replica sets), three configuration (config) servers, and one or more query routers. The shards are spread over different data nodes, though multiple shards can co-exist on a single machine. The full collection of shards in the database is managed by three config servers. The config servers store the cluster's metadata, i.e., they keep track of the mapping of the cluster's data to the various shards in the database. For fault-tolerance, having three config servers is recommended, to deal with hardware failures. A cluster with a single server can be fully operational. Finally, query routers are tasked with analysing queries and rerouting queries to the appropriate shards. Having many query routers is recommended, as all queries must pass through a query router, thus having too few query routers can cause this step in execution to become a bottleneck.

2.2 The Hadoop Distributed File System (HDFS)

HDFS is the distributed file system that is the underlying architecture of the well-known MapReduce framework Hadoop. In contrast to the database structure that MongoDB uses, HDFS deploys a master/slave architecture that allows for file-system-like operations, e.g., opening, closing, and renaming files. Every file is split up in blocks and every block is replicated a configurable amount of times. The replication process, including the placement of the replicas, is configurable, although a default policy exists that replicates each block three times. In this policy one replica is placed on the local node of the writer and two more are placed on different nodes in one remote rack. Whereas this policy does not spread the replicas evenly over different racks, it does reduce network bandwidth usage since only two rather than three different racks are used. In addition it maintains both read performance and data reliability, since the chance of rack failure is far smaller than that of node failure.

A typical HDFS deployment consists of a single NameNode and a number of DataNodes, usually one DataNode per node in a cluster. The NameNode takes the role of master and is responsible for the regulation of client access to the files as well as the maintenance of the file system namespace. The DataNodes on the other hand manage the storage that is attached to the node they are running on. They must then allow clients to actually perform the requested read/write operations. In addition, they take instructions from the NameNode with respect to block removal and duplication.

2.3 VMware

VMware is a software company that provides software for cloud management and virtual machine management. VMware offers a large virtualization ecosystem with many layers of applications. At the core is the hypervisor, known as VMware ESXi, which is responsible for running virtual machines on physical hardware. Multiple ESX hosts can be combined into a network that is managed by a vCenter instance. Each network managed by such a vCenter instance is known as a Virtual Data Center (VDC). VDCs can be combined to form the vCloud, the overarching system responsible for managing all VMs in a company's private cloud. Bitbrains has deployed a vCloud across their three clusters, which are situated in three physical data centres. They have grouped their servers into two VDCs, whereby each VDC consists of servers spread over all three of Bitbrains' physical clusters. This setup has a greatly reduced chance of a VDC becoming unavailable compared to a setup of one VDC per physical cluster.

2.4 OpenStack

Similarly to VMware, OpenStack also offers software that allows for virtualization management. This open-source variant deploys a modular architecture, with Nova, or OpenStack compute, at its core. Whereas Nova is responsible for the resource pools, other modules such as Swift and Neutron have other responsibilities (storage and networking respectively). The OpenStack API has also been made compatible with the well-known cloud provider Amazon's Elastic Compute Cloud (EC2) [6]. Whereas Bitbrains has no deployment of OpenStack and due to their VMware-certified status will have no such deployment for the foreseeable future, the choice for OpenStack comes from the wide use in the rest of the industry as well as it's compatibility with Amazon's EC2.

3 Existing Solutions

In this section, we analyse some existing for virtualization awareness (Section 3.1), deploying applications on a set of resources (Section 3.2), and testing the influence of our solution on various distributed applications (Section 3.3).

3.1 Virtualization Awareness

To the best of our knowledge, only one solution for providing virtualization awareness to distributed applications exists. This solution is further described in this section.

Hadoop Virtualization Extensions To facilitate deploying Hadoop on top of VMware-based clouds, VMware has developed software known as VMware Big Data Extensions as described in a white paper [7]. The software provides Hadoop with information about the physical architecture underlying the virtual machines the application is running on. To achieve this, VMware has extended the popular Hadoop MapReduce framework with the Hadoop Virtualization Extensions to utilise the mapping of virtual to physical machines. This extension adds a single layer to the Hadoop network topology that describes the physical nodes VMs are deployed on. In addition, VMware has extended the block placement policy in HDFS to prevent it from placing multiple replicas of the same block on a single physical host.

VMware has made available an open source variant of their Big Data Extensions, known as Project Serengeti. This project is strongly coupled with both VMware and Hadoop, which limits its use as a generic solution for virtualization awareness. As Serengeti was not designed with additional applications or virtual machine managers in mind, extending the project would likely be time-consuming and require substantial changes to existing code to create a simple, generic interface. Project Serengeti also lacks support for vCloud, a part of the VMware software stack used extensively at Bitbrains. As a result, the parts of Serengeti related to VMware will not be used during our project. The Hadoop extensions, however, will be reused. These extensions are part of the Apache Hadoop distribution as of version 2.2.

Table 1: State-of-the-art MapReduce benchmarks and use cases.

	Queries/Jobs	Workload Diversity	Data Set	Data Layout	Data Volume
MRBench [9]	business queries	high	TPC-H	relational data	3 GB
N-body Shop [10]	filter and correlate data	reduced	N-body simulations	relational data	50 TB
DisCo [11]	co-clustering	reduced	Netflix [12]	adjacency matrix	100 GB
MadLINQ [13]	matrix algorithms	reduced	Netflix [12]	matrix	2 GB
ClueWeb09 [14]	web search	reduced	Wikipedia	html	25 TB
GridMix [15], PigMix [16]	artificial	reduced	random	binary/text	variable
HiBench [17], PUMA [18]	text/web analysis	high	Wikipedia	binary/text/html	variable
WL Suites [19]	production traces	high	-	-	-
BTWorld [20]	P2P analysis	high	BitTorrent logs	relational data	14 TB

3.2 Resource Management

Resource managers for distributed applications are more widespread, with Apache Mesos and YARN being popular choices. The applicability of both of these resource managers is discussed in this section.

Mesos Apache Mesos [8] presents itself as a platform for sharing clusters between multiple distributed computing frameworks. Mesos requires computing resources to be assigned to it, so that it can deploy distributed applications on these resources. It will then offer these resources to the various applications that are deployed by Mesos, and these applications are able to either accept or reject the resources they are offered.

Although Mesos is flexible in its resource allocation, it requires support from the deployed applications to function. In addition, Mesos can not contact a virtual machine manager to launch additional virtual machines for use by applications. Mesos also lacks virtualization awareness, and as a result the information on physical locations is not available to deployed applications. Adding these missing features to a system as complex as Mesos requires a significant amount of redesigning and refactoring. This has led to the decision that designing a virtualization aware system from scratch will require less time and leads to a better design.

YARN YARN is another resource manager by Apache. YARN is part of the Hadoop project, and as such it is virtualization aware through the use of the Hadoop Virtualization Extensions. YARN does require an external program to provide it with topology information (a mapping from machines to locations in the physical topology). Like Mesos, YARN does not obtain its own resources. Adding machines to a YARN setup requires starting a YARN service on every machine. Another limitation of YARN is that it requires the applications it deploys to run in a YARN "container". This means that each application has to be designed specifically for YARN and must be written in Java. For this project, we will support YARN/Hadoop as an application and use its resource management to run YARN-enabled applications with virtualization-awareness.

3.3 Testing Distributed Systems

In previous work [20], we have compared state-of-the-art MapReduce workloads and benchmarks for properties such as workload diversity, data volume, and application. When testing Hadoop, these workloads can be used to simulate the load that Hadoop may experience in a production environment. The characteristics of several state-of-the-art workloads are summarised in Table 1 (taken from previous work).

Of particular interest to this project are the high diversity workloads. Customers at Bitbrains have many different queries that they need to run on their data, and as a result a benchmark focusing on a single aspect can not cover the needs of all customers. By testing with high diversity workloads, we are more likely to cover the different aspects of the workloads that Bitbrains faces. From the comparison made in Table 1 it follows that the most relevant (and easiest to access) workloads are thus the HiBench [17]/PUMA [18], WL Suites [19] and BTWorld [20] workloads. For testing we will be using both HiBench and BTWorld, excluding others due to time constraints and similarities with the two we have chosen.

For MongoDB, benchmarking options are not as well explored and documented. One of the only widespread benchmarks for MongoDB is the Yahoo Cloud Serving Benchmark [21]. The benchmark con-

Table 2: Basic information about the employees interviewed for this project.

Name	Function description	Years of employment
Jeroen van Nieuwenhuizen	Engineer, Hadoop deployments	1
Bas Welman	Platform engineer, handles everything up to VM level	3
Gjalt van Rutten	CTO, deploying and implementing the technical vision	Since the start

sists of a data generator and a set of workload descriptions based on common use cases for databases like MongoDB. Based on these descriptions, common workloads can be simulated for various database sizes.

4 Requirements

For the process of requirements gathering we have mainly focused on interviewing Bitbrains employees with varying specialities, since they are best equipped to inform us of how Bitbrains runs its operations and what features in our platform they would benefit from most. The resulting requirements and the main results of these interviews will be described in this section as follows: we summarise the results of these interviews in Section 4.1 and give a list of the requirements in Section 4.2.

4.1 Interviews

To get a clear view of the way Bitbrains is currently deploying their distributed applications on their infrastructure and what they would like to achieve, we have spoken with three of their employees, in addition to our external supervisor. The basic information about each of these employees in summarised in Table 2.

Through our interview with Jeroen we learned that Bitbrains offers virtual machines in both a Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) manner, though so far only PaaS has been used by customers. For their PaaS products Bitbrains is responsible for the configuration of the software running on the virtual machines. Every customer gets their own separate configuration, manually installed by Bitbrains engineers. Any virtual machines used for Hadoop can not be easily physically relocated, since they are linked to the physical hard drive they are working with. When asked whether Bitbrains had any workloads or traces we could use to see if our system improves over the current system, Jeroen informed us that they have used TeraSort in the past. He would however also be interested in having access to traces or workload specifications, so he is going to contact customers about the possibilities thereof. There are currently few MongoDB deployments in place, which means that for MongoDB there are currently no workloads available from Bitbrains.

The interview with Bas focused more on the infrastructure in place at Bitbrains, from the connection between different data centres to the connection between CPU and storage within the same rack. The most relevant item discussed during this interview is the VMware deployment Bitbrains currently has. The deployment runs an instance of vCloud on top of two instances of vCenter. Due to limitations imposed by the implementation of vCenter, all data centres have at least one rack that belongs to the first vCenter and at least one that belongs to the second. They are currently running vCenter 5.1, but will be upgrading to 5.5.1 as soon as it is available. As a result, compatibility should be considered for the implementation of the VMware extension.

Our final interview, with the CTO of the company Gjalt van Rutten, was perhaps the most illustrating interview. Whereas the interview with Jeroen had left us with the impression that moving virtual machines would be unwanted behaviour, Gjalt informed us that it is definitely possible, but just a costly operation. In addition, Hadoop could also be deployed to nodes with shared storage, which would allow rescheduling of the computational nodes while keeping the storage location static. This opened up the project to be more similar to the initial project idea, including scheduling decisions of VM placement. Gjalt was rather clear on the kind of system he would like to see for Bitbrains, with users being allowed to specify policies for deployment of the VMs and the management thereafter.

4.2 Requirements

The middleware is located between the VM manager and the distributed applications that run on the virtual machines. The APIs that are provided to both sides allow the middleware to perform three main tasks.

The first of these tasks is virtual machine deployment. In this phase, a user wants to set up a specified number of VMs to run a distributed application. By specifying the user's priorities in terms of fault tolerance and performance, the middleware layer can provide suggestions to the VM manager on how the total number of virtual machines should be distributed over the available physical data centres.

The second task is the scheduling of virtual machines at runtime. When virtual machines are placed in a cloud, the middleware is able to relocate VMs within the cloud to use different physical CPUs or storage units. This relocation is done based on numerous policies which can be specified by the application.

The last task of the middleware is the presentation of parts of the physical hardware topology to the distributed application. By showing applications on what physical hardware its VMs are running, the applications can select a better relocation policy or modify internal behaviour to improve performance, provide sufficient replication, etc.

Below we present an initial list of formal requirements for the middleware layer. Additionally, we provide an initial list of formal requirements for the MongoDB extension. This extension will be used to demo the potential of our final product.

Deployment

- Middleware should accept user provided policies that specify how VMs should be placed during deployment.

Scheduling

- Middleware can relocate virtual machines at runtime based on a number of policies which can be selected by the distributed application.

Topology

- Middleware should retrieve topology changes from the VM manager when these occur.
- Middleware should communicate topology changes to distributed applications when these are detected.
- Middleware should inform distributed applications which of its virtual machines are running on the same physical CPU and/or storage units.

MongoDB

- Data replication is set up automatically.
- Primary replicas are placed on separate physical disks to ensure load balancing during write-intensive operations.

5 Approach

In this section we will describe our approach to the project, starting with the development methodology as described in Section 5.1. In Section 5.2 we present a broad overview of our system design. An initial version of the API of our application is introduced Section 5.3. Finally, the programming languages and tools we will use for the project are specified in Sections 5.4 and 5.5 respectively.

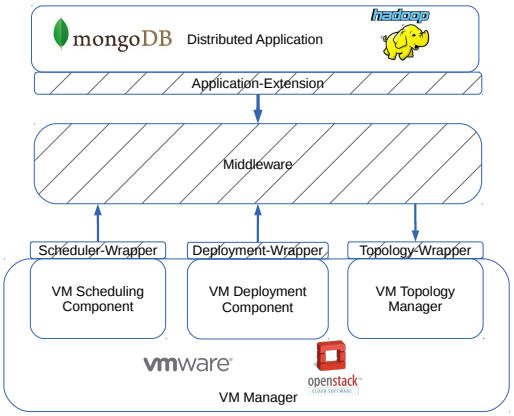


Figure 1: A broad overview of the system architecture.

5.1 Development Methodology

During the software development phase, the team will be applying agile development methods. In particular, the Scrum development framework will be used. Alternatives we have considered include the waterfall methodology and XP, extreme programming.

We have opted not to use the waterfall method due to its rigid and inflexible nature. For this project we foresee having to cope with design changes, which is difficult and time-consuming once implementation has started in the waterfall method. Throughout the project we will be designing multiple APIs, i.e., the interface we provide to distributed applications and the interface we require from VM managers. In practice, the design of an API often has to go through multiple revisions to accommodate new requirements, incorporate new insights in the problem, etc. For example, Google recommends writing an API draft of at most two pages and improving on it in iterations.

On the other hand, agile methods feature an iterative approach to software development, and are flexible enough to alter the design of a product throughout a project. As a result, using an agile method enables us to focus on the core of the product during initial stages, and add more functionality as the project progresses. It also allows us to keep a list of features that are not required for a functional product, but would increase its usefulness if implemented. These features can be prioritised based on the client's demands to ensure the final product contains the most requested and most valuable features that could be implemented in the limited timeframe.

Our choice for Scrum over XP as the preferred agile method is mainly based on experience of the team working with Scrum. On top of this, extreme programming is less suitable for small teams due to its strict requirements on code reviewing and testing, i.e., using pair programming. Due to the limited time available for this project, we prefer being able to work in parallel to ensure we cover more features. Section 6 contains more detail on how we will ensure sufficient quality using our chosen methodology.

5.2 System overview

Our software will be placed in an existing system with two major, relevant components or domains: the distributed application and the VM manager. To achieve the abstraction between distributed applications and VM managers, our system will add extensions to both sides. In our initial design we considered designing a single API for communication between the two domains. Scheduling and deployment would have been implemented in the VM manager extension. One of the main advantages of this design is the possibility to

integrate all of the software we write into existing software, that is, there is no external application that needs to run on some machine. Another advantage is that there is only a single layer of additional communication. The major disadvantage of the extension-only approach is that different VM manager extensions will have significant overlap in the deployment and scheduling implementation. Sharing code between extensions will result in an unmaintainable project, and thus the design does not allow for easy extension to additional VM managers.

After several iterations our system design still has one extension per domain (distributed or VM manager), but adds a middleware layer between the two domains as illustrated in Figure 1. Both distributed applications and VM managers will communicate with the middleware layer through APIs, as discussed in Section 5.3. The middleware contains a scheduler that places VMs based on user-specified placement policies. In addition, the middleware retrieves topology information from the VM manager and forwards this data to the distributed application. One advantage of this design is the possibility for common features, such as the scheduler, to be implemented independent of the underlying VM manager. Additionally, in our design the distributed application and VM manager do not communicate directly, which reduces security risks for the cloud operator. A downside to the middleware approach is having to design and maintain multiple APIs. This could also be interpreted as an advantage; changing one of the APIs has impact on fewer extensions, thus adding features to the software requires a smaller investment.

5.3 Initial API

Since we will develop our product in an iterative manner, no full system specification and design has been thought up yet. As recommend by Google in their lecture on designing APIs [22], we instead present an initial API design here, which will be implemented during the first sprint(s) of this project. During this development we will continuously change this API to reflect the latest wishes of the customer, thus keeping the development process highly flexible. In this section of the report we will give a broad overview of the structure we have chosen for our software in Section 5.2 as well as the initial version of the API in Section 5.3. We also specify which programming languages and tools we will use in Sections 5.4 and 5.5 respectively.

The API our middleware will expose to the distributed applications will take the form of a REST API [23]. A REST API is a generic format commonly used in web-based services that should satisfy the following requirements:

- A base Universal Resource Identifier (URI) should be defined through which the API can be accessed.
- The data should be returned in some Internet format, such as JSON or XML.
- The following four HTTP methods should be implemented by the API: GET, PUT, POST and DELETE. These methods are for retrieving, storing, updating and deleting data respectively.
- The return data should contain URIs or part thereof that can be used to access further information where appropriate.

As many of the current services and public APIs are RESTful, implementing this standard will provide a framework users of our API will already be familiar with. In addition, two of our group members have experience with the implementation of such an API from an earlier project at the Delft University of Technology.

As described in Section 4.2, our middleware provides three main services: 1. virtual machine placement during setup, 2. virtual machine scheduling during run time, and 3. providing physical topology information to distributed applications. The initial version of the API provides basic functionality for each of these three services. For the placement of virtual machines a method will be provided that takes a placement policy and a number of VMs. It returns a distribution of VMs over available physical data centres. The scheduling of virtual machines at run time will enable distributed applications to request its VMs to be placed either on the same or different physical machines. The topology of the physical infrastructure will be available to distributed applications. Applications can use this information to create a map for each of its VMs to a physical machine. The full initial version of the API is described in Appendix A.

Table 3: Libraries and development and process tools chosen for this project.

Tool	Purpose
BibTeX	References in LaTeX documents.
Doxygen	Documentation of non-Java languages.
EclEmma	Visualisation of test coverage.
Eclipse	Implementation in Java.
Git	Version control of code and documentation.
Guice	Allows for a cleaner alternative to the factory-model.
JavaDoc	Documentation of Java code.
Jenkins	Continuous integration.
Jersey	RESTful client/server in Java.
JIRA	Scrum-planning tool
JUnit	Testing of Java code.
Jukito	Combination of JUnit, Mockito and Guice.
LaTeX	Process documentation.
Mockito	Mocking of objects for testing.
Vim	Implementation in other languages (e.g. C++).

5.4 Programming languages & Libraries

Because our software is used by distributed applications and virtual machine hypervisors, which can run on numerous different platforms, code portability is an important factor. To this end, we choose Java as the main language for our code. Java runs on the Java Virtual Machine and is portable between Linux, Windows, OS X, and other systems. Initially, the product will work with MongoDB and VMware. VMware provides a Java API by default, which is an additional reason to start with Java. In general, however, distributed applications and VM hypervisors can be written in any language. We use the native language of the application or hypervisor for its respective extension. In the case of MongoDB, this will be C++.

Using existing libraries and frameworks can save time and effort during the development of a software product. Because our product will include an extension for VMware, we use of the VMware vSphere Java API. This API takes care of networking and allows the programmer to retrieve information about the physical network and the virtual infrastructure. For portability reasons, communication with the middleware layer will happen through the use of RESTful APIs. More specifically, we will use the Jersey library for both the client and server connections in our applications. Not only is Jersey quite commonly used, it is also the reference implementation of JAX-RS (Java API for RESTful Web Services) provided by Oracle. Since we have no previous experience with any Java-based REST client/server and a lot of support and documentation is available for Jersey, we have chosen to use this library.

5.5 Tools usage

In order to develop a product of this size, both development and process tools can help to keep an overview of the tasks at hand. In order to easily keep track of the increasing number of lines of code, a good IDE and some version control software can be very insightful. Similarly planning tools such as the agile-orientated JIRA can help to provide a clear overview of the current ToDo-list. In Section 5.5.1 we describe the development tools we will use in this project, whereas Section 5.5.2 will focus on the process tools. A summary of the tools and libraries we will use is presented in Table 3, and a list of all tools and libraries mentioned in this report, including their website is presented in Table 4.

Table 4: The tools and libraries mentioned in this report and their website.

Tool	Website
BibTeX	http://www.bibtex.org
CircleCI	https://circleci.com/
Doxygen	http://www.stack.nl/~dimitri/doxygen/
EasyMock	http://easymock.org/
EclEmma	http://www.eclemma.org/
Eclipse	https://www.eclipse.org/
Git	http://git-scm.com
Guice	https://code.google.com/p/google-guice/
JavaDoc	http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html
Jenkins	http://jenkins-ci.org/
Jersey	https://jersey.java.net/
JIRA	https://www.atlassian.com/software/jira
JUnit	http://junit.org/
Jukito	http://jukito.arcbees.com/
LaTeX	http://www.latex-project.org/
Mockito	https://code.google.com/p/mockito/
NetBeans	https://netbeans.org
Planbox	https://www.planbox.com
SVN	http://subversion.apache.org/
TestNG	http://testng.org
Vim	http://www.vim.org

5.5.1 Development tools

As has been previously described in Section 5.4, we will develop the largest part of our code in Java. For this part of the code, we will use the Eclipse IDE. Eclipse is both commonly used for Java and is an IDE we have experience with. The alternative in the form of NetBeans has been considered, but since we have experience with Eclipse, we believe this will allow for a smoother workflow and switching to the similar but slightly different NetBeans would hold no advantages. In contrast to an extensive IDE such as Eclipse, the alternative of Vim has also been considered. Vim is a command-line text editor that offers many development features for those that are used to its somewhat peculiar set of shortcuts. The downside is that for large projects it is quite difficult to keep a clear overview of all code. For small extensions such as that to MongoDB that will not be written in Java, Vim will be used.

In addition to an IDE, we will also require version control to easily keep track of code changes. Version control systems manage changes in code and/or documents, and provide a central location to store the code and documents. We choose Git as our version control tool. Our project team has experience with both Git and SVN, and both version control systems are commonly used in academia and industry. As we are using an agile development method, a working version of the product, preferably with new features, should be presentable at the end of every sprint. To maintain a stable version of the product, branched development can be a huge benefit. This allows for a main branch that contains the proven to be stable version of the software, with new features being developed in separate branches. Because Git has built-in support for branch-based development, it is preferred over SVN which offers a very primitive branching system which involves manually creating the folders for branches and merging them afterwards.

Given that the core of our application is written in Java, many testing frameworks are available. Out of the different options the team has the most experience with JUnit, as this was also the basis of the Software Quality & Testing course. TestNG has slightly more features, such as group-based testing. We still choose for JUnit due to it's native support in Eclipse and our previous positive experience with this framework. We

will however extend JUnit with use of Mockito that allows for use of mocked objects that can for instance take the place of files in testing. Not only is Mockito more advanced than some of the other frameworks that allow for mocking, such as EasyMock, the team also has experience with Mockito through the aforementioned course. In addition the usage of Guice, a Google library that allows for so-called injections. This allows one to write cleaner tests and production code, by eliminating the factory-model. Jukito is a library that combines JUnit, Mockito and Guice which sounds very promising and will be tried by the team. Finally we will use an Eclipse plug-in called EclEmma that allows one to easily inspect what code and what branches have been tested. Other tools exist for this, but this plug-in allows one to see the results either as an overlay in the editor, or as a separately generated directory of HTML files, which is sufficient for us to keep an overview of the test coverage.

5.5.2 Process Tools

In addition to the tools required for the development, we are also using a planning tool to keep track of our planning. JIRA is an online tool that allows for an overview of the current work, as well as some scrum-oriented features. One of these features is the scrum board which mimics the structure of ToDo, Work in Progress, Done that is often done with sticky notes on a white board. Though an analogue version of this system, might be more intuitive and more easily accessible, it does require us to be at the same physical location every day in order to access the board. Since our time is divided between Bitbrains and the Delft University of Technology, a digital version is more practical for us. Planbox is an alternative to JIRA that we have previously used for a university project and has been considered for this project as well. It too offers support for agile development methods, with support for iterations in the form of sprints combined with a backlog. However, we have ultimately chosen for the use of JIRA, since Bitbrains is also using this tool. This allows our supervisor to also have easy access to our planning tool.

6 Quality Guarantees

In this section we talk about the quality of both the process and the code. In Section 6.1 we talk about the type of documentation we use in the code and why. In Section 6.2 we talk about the way we write and run tests. In Section 6.3 we will discuss the evaluation of the quality of our code, and how we guarantee the quality of our reports and documents.

6.1 Documentation

Two different types of documents will be required for this project, the first being documentation of the process, including this document, the second being code documentation. All documents related to the process will be produced in LaTeX, supplemented with BibTeX when references are a part of the report. For these documents the version control system Git will also be used.

For our project, we choose to use JavaDoc and Doxygen for documentation. The reason for using JavaDoc is that it is the de facto standard for documentation generation in Java. Doxygen is one of the most popular documentation generation tools available for C++ and numerous other programming languages like C and Python. Our product relies heavily on extensions for both the supported applications and the virtual machine hypervisors. This software can be written in any language, and therefore requires a flexible documentation generation tool. We choose either the standard deployed by the software-package or Doxygen if there is no standard, to generate documentation for the extensions.

6.2 Testing

As new functionalities are added to a product, it is always important to verify that existing functionality is not broken during the latest update. To this end a test suite containing both unit tests an integration tests can be a great help. To evaluate our code, we use test-driven development. With test-driven development,

unit-tests are written before the code that should be tested. This allows us to specify the working of our code through the tests that are initially based on an API. These tests often take the form of user stories: "I am X and if I do Y, then Z will happen". The use of a test-driven development approach leads to spending less time on debugging code, and to more modular and extensible code in general. Additionally, we want our code to be well-thought-out and ensure that all team members have a large understanding of the code. To this end, the unit-tests for a certain part of the product code are written by a different team member than the actual product code.

Simply writing the tests is of course not sufficient for them to be useful, since tests also need to be run to get any data out of it. A continuous integration server with hooks into Git allows you to have the tests run automatically on every push to the server. We will use this hook to run our unit tests using mocks after every push. Jenkins is a commonly used example of such a continuous integration platform and is also the one we will be using for this project. Though we have some experience with CircleCI as well, Jenkins is more commonly used and provides all the required features. Testing in a live environment will be done at the end of every sprint, outside of the continuous integration tools.

6.3 Evaluation

Because we want to deliver a high-quality product to our customer, we evaluate both our code and documents carefully before delivery. To this end, we use multiple methods.

All documents we deliver are written using an iterative method. This means all sections are both read and edited multiple times by more than one team member. This method has the following benefits:

- All team members are aware of document contents.
- Section contents are refined and reflect the accurate opinion of the team.

At the end of the development process, we use benchmarking techniques to evaluate the performance of our software. Because Bitbrains provides IaaS and PaaS services to other commercial companies, obtaining real-world representative workloads is a non-trivial task. The performance evaluation will consist of running real-world workloads, if these can be obtained from Bitbrains' customers, or using benchmarking tools like TeraSort and Yahoo Cloud Serving Benchmark for Hadoop and MongoDB respectively.

References

[1] MongoDB, inc, "MongoDB." `http://www.mongodb.com`, April 30 2014.

[2] VMWare, inc, "VMWare." `http://www.vmware.com`, April 30 2014.

[3] The Apache Software Foundation, "Welcome to Apache Hadoop." `http://hadoop.apache.org/`, April 30 2014.

[4] OpenStack, "OpenStack: Open Source Cloud Computing Software." `https://www.openstack.org`, April 30 2014.

[5] MongoDB, "Datacenter Awareness." `https://www.youtube.com/watch?v=1XmizTfw5a8`, October 4 2012.

[6] Amazon Web Services, inc, "Amazon Elastic Compute Cloud (EC2)." `https://aws.amazon.com/ec2/`, April 30 2014.

[7] VMWare, inc, "Hadoop Virtualization Extensions on VMware vSphere 5." `http://www.vmware.com`, April 23 2014.

[8] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pp. 22–22, 2011.

[9] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, "MRBench: A Benchmark for MapReduce Framework," in *ICPADS*, 2008.

[10] S. Loebman, D. Nunley, Y.-C. Kwon, B. Howe, M. Balazinska, and J. P. Gardner, "Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help?," in *Cluster*, pp. 1–10, IEEE, 2009.

[11] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *ICDM*, pp. 512–521, IEEE, 2008.

[12] "Netflix prize."

[13] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "MadLINQ: large-scale distributed matrix computation for the cloud," in *EuroSys*, pp. 197–210, ACM, 2012.

[14] "The ClueWeb09 Dataset."

[15] "The GridMix Hadoop Benchmark."

[16] "The PigMix benchmark."

[17] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The Hibench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *ICDEW*, pp. 41–51, IEEE, 2010.

[18] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "PUMA: Purdue MapReduce Benchmarks Suite," tech. rep.

[19] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance Using Workload Suites," in *MASCOTS*, pp. 390–399, IEEE, 2011.

[20] T. Hegeman, B. Ghit, M. Capota, J. Hidders, D. Epema, and A. Iosup, "The btworld use case for big data analytics: Description, mapreduce logical workflow, and empirical evaluation," in *Big Data, 2013 IEEE International Conference on*, pp. 622–630, IEEE, 2013.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[22] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 506–507, ACM, 2006.

[23] M. Masse, *REST API design rulebook*. O'Reilly Media, Inc., 2011.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

A Preliminary Design

A.1 User-Stories

The main requirements that a user of the system will actually notice are most clearly presented in so-called user stories. These stories describe what the results of an action will be for a certain user. Unfortunately many of the other requirements or design decisions can not be represented in the same format that easily, since they do not really involve a user in the classic sense of the word. Instead some of the functions that the distributed application should be able to use have been described in scenario format.

The user-stories that describe the behaviour as presented to the user can be summarised as follows:

Name	User-Policy
I am	a user,
When	I start deployment,
Then	I can specify policies for VM placement before and after deployment.
Name	User-Deployment
I am	a user,
When	I start deployment,
Then	the system will advise me on which clusters to use.
Name	User-Running
I am	a user,
When	my system has been deployment,
Then	the system will relocate my tasks over compute and storage nodes within the same cluster.
Name	Application-Transfer
I am	an application,
When	I am deployed,
Then	I can deny and request VM transfer through the policy specified by the user.

Whereas more scenarios will be developed as new features are picked to be implemented, a few scenarios that result in the API described in Appendix A.3 are given here:

Given	a MongoDB deployment,
When	I request the collection all my VMs,
Then	I get a list of virtual UUIDs.
Given	a MongoDB deployment,
When	I request the information of an running VM,
Then	I get the physical and virtual UUID as well as the current performance statistics.
Given	a MongoDB deployment,
When	I request the information of a non-existing VM,
Then	I get an error response indicating this is not possible.
Given	a MongoDB deployment,
When	I request the information of an existing physical machine,
Then	I get a list of VMs running on this machine.
Given	a MongoDB deployment,
When	I request the information of a non-existing physical machine,
Then	I get an error response indicating this is not possible.
Given	a MongoDB deployment,
When	I require a number of VMs combined with requirements in the form of policy,
Then	I get a list of VMs fulfilling these requirements.

A.2 Generalised Topology

One of the tasks of the middleware is to generalise the physical topology to ensure that the distributed applications get the same topology format regardless of the VM manager that is running. The generalised topology we have chosen matches that of HDFS to a certain extent, taking the form of a tree. The root of the tree represents the cloud, with data centres as its children. Each data centre contains many racks and every rack houses computation, storage and possibly combined nodes. This topology tree is visually represented in Figure 2.

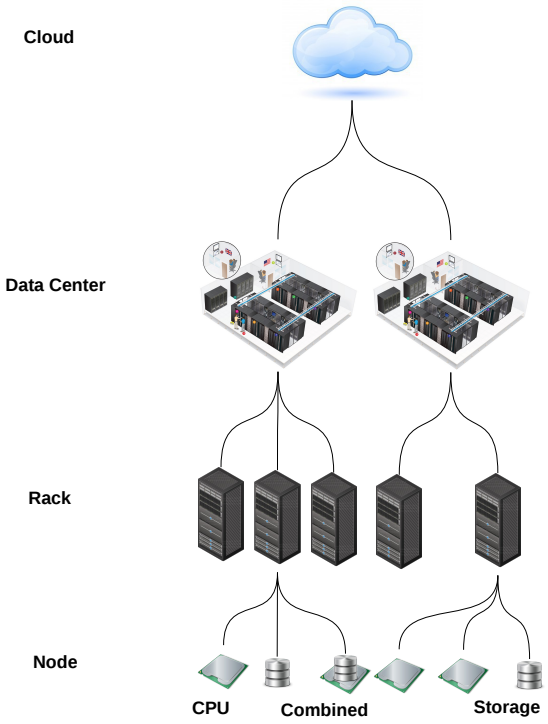


Figure 2: The generalised topology our middleware will use.

A.3 API

The initial version of the API that is able to fulfil all scenarios described in Appendix A.1 is described in Tables 5 and 6. The codes referred to in the last column of both tables are the traditional HTTP status codes [24].

Table 5: API exposed by the middleware to the distributed applications.

Type	URL	Parameters	Request Body	Response		Codes
GET	/vms/	-	-	vUUIDs	[String]	200 500
GET	/vms/:vUUID/	-	-	pUUID vUUID cpu_load io_load mem_load net_load	String String Integer Integer Integer Integer	200 404 500
GET	/phys/pUUID/	-	-	vUUIDs	[String]	200 404 500
GET	/deployment/	-	-	vUUIDs	[String]	200 500

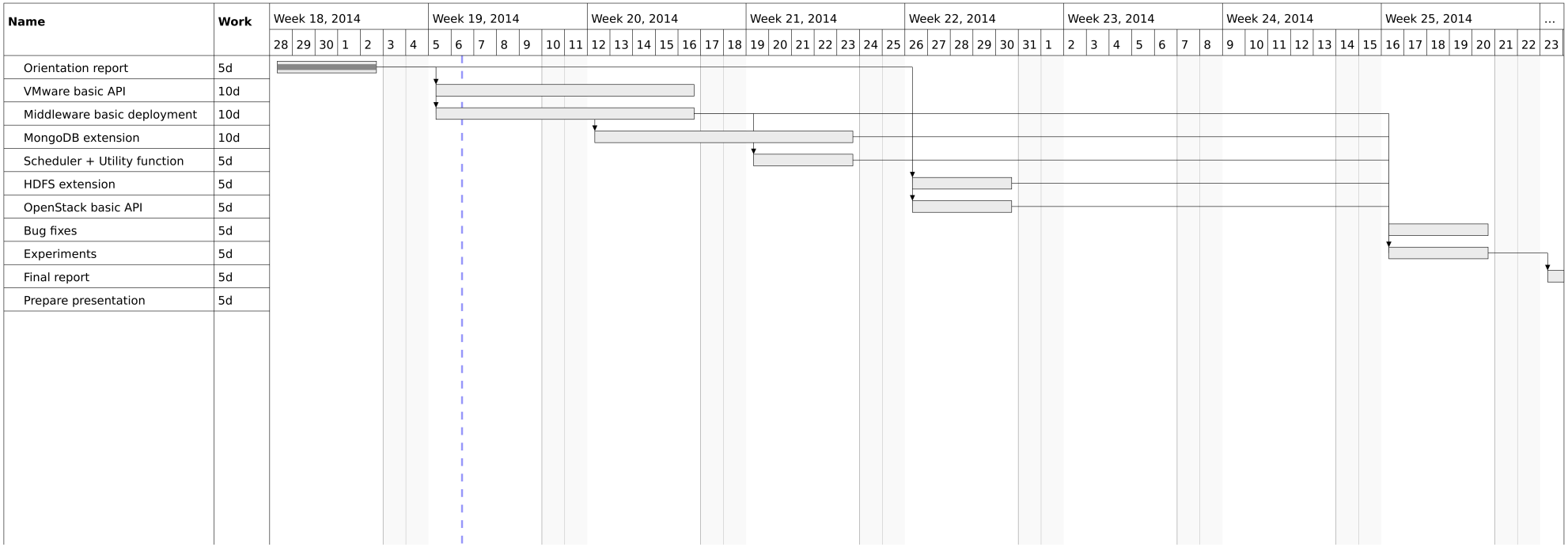
Table 6: API exposed by the VM-manager extensions to the middleware.

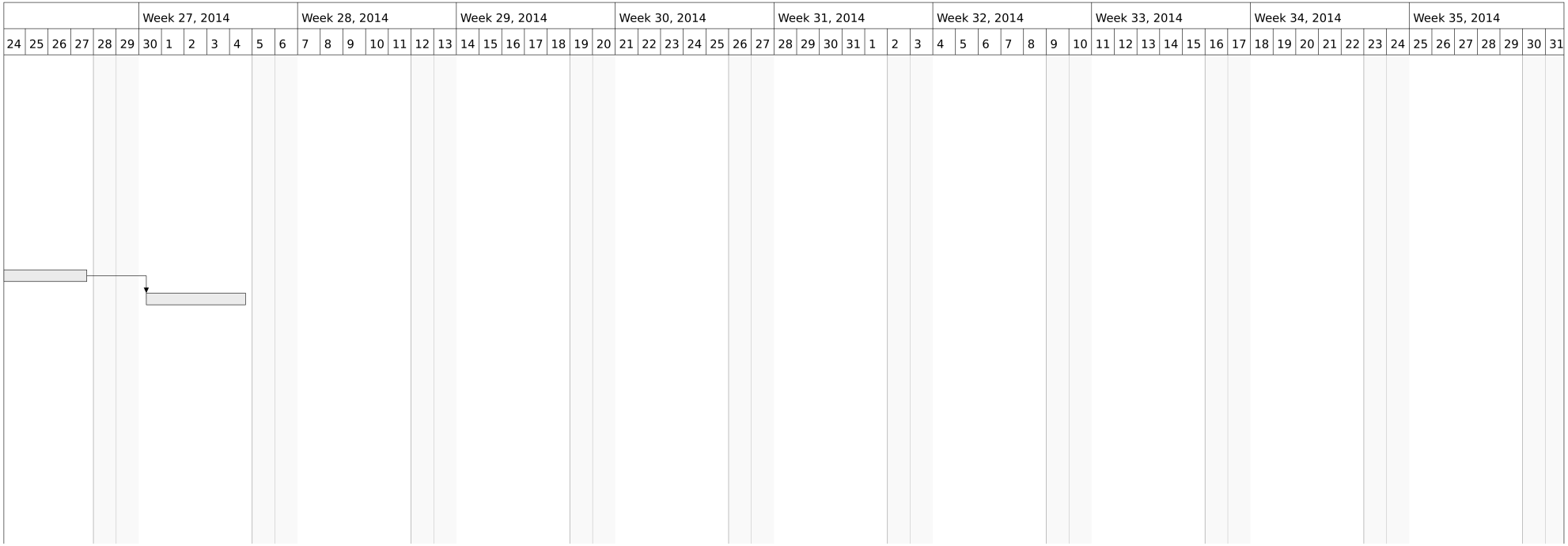
Type	URL	Parameters	Request Body	Response		Codes
GET	/vms/	-	-	vUUIDs	[String]	200 500
GET	/vms/vUUID/	-	-	pUUID vUUID	String String	200 404 500
GET	/topology/	-	-	topology	Tree-Object	200 500

B Planning

Even though we will use the agile scrum methodology to continuously update the planning, an initial planning with some important dates has been made. The implementation phase will start 5th May 2014. No new features will be implemented after 16th June 2014. The entirety of the initial planning has been summarised in a Gantt chart on the next pages. Week 23 and 24 have been left empty on purpose. This time will primarily be spent on incorporating additional features and requirements that could be added in later sprints. It also allows for troubleshooting of unforeseen general issues.

Name	Work
Orientation report	5d
VMware basic API	10d
Middleware basic deployment	10d
MongoDB extension	10d
Scheduler + Utility function	5d
HDFS extension	5d
OpenStack basic API	5d
Bug fixes	5d
Experiments	5d
Final report	5d
Prepare presentation	5d





B

SIG Code Evaluation

B.1. Initial Evaluation

[Aanbevelingen] De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Duplication en Module Coupling.

Voor Duplicatie wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren. In dit systeem is er bijvoorbeeld duplicatie te vinden tussen de 'topologyManager'-classen die geimplementeerd zijn binnen de componenten 'Nebu-app-hadoop'[1] en 'Nebu-app-mongo'. Ook tussen de verschillende 'Builder'-classen is duplicatie te vinden. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Voor Module Coupling wordt er gekeken naar het percentage van de code wat relatief vaak wordt aangeroepen. Normaal gesproken zorgt code die vaak aangeroepen wordt voor een minder stabiel systeem omdat veranderingen binnen dit type code kan leiden tot aanpassingen op veel verschillende plaatsen. In dit systeem wordt de class 'VirtualMachine' op ruim 56 verschillende plaatsen aangeroepen. Daarnaast is deze class vrij fors. Het lijkt erop alsof deze class twee verschillende type functionaliteit bevat, het is een representatie van een 'VirtualMachine' en bevat de code voor een builder van dit object. Om zowel de grootte als het aantal aanroepen te verminderen zouden deze functionaliteiten gescheiden kunnen worden, wat er ook toe zou leiden dat de afzonderlijke functionaliteiten makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden worden.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

B.2. Final Evaluation

[Hermeting] In de tweede upload zien we dat zowel de omvang van het systeem als de score voor onderhoudbaarheid licht is gestegen. Wat betreft de Duplicatie zien we een flink afname in gedupliceerde code in het C++ gedeelte van het project. De introductie van het 'Nebu-app-framework' lijkt dan ook geholpen te hebben om de gezamenlijke functionaliteit te centraliseren. Ook voor de Module Coupling zien we een stijging in de score, het afsplitsen van de verschillende 'Builder' classen lijkt ook hier een positieve invloed te hebben gehad. Als laatste zien we bij zowel het C++ als het Java gedeelte niet alleen een stijging in de hoeveelheid productie code, maar ook een stijging in de hoeveelheid test-code.

[1] This message has been redacted to replace the work-in-progress name of the project with the name of the final project

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie goed zijn meegenomen in het ontwikkeltraject.

C

Original Project Description

Bitbrains is a service provider that specialises in enterprise level managed hosting and financial risk calculation. The majority of the business comes from the banking and insurance industry with customers like: ING, Nationale-Nederlanden, Aegon, Ahold, and other large enterprises. Bitbrains uses VMware virtualization to manage the large multi tenant environment that hosts all the customer services. Some of the software packages used by Bitbrains customers depend on distributed data storage. One of the systems used for this is MongoDB. Most of the currently available distributed data-platforms like Hadoop and MongoDB are not designed with Infrastructure Virtualization in mind. The problem with this is that these platforms can not handle situation where multiple Virtual Machines (VMs) are running on the same Physical server. For Hadoop in general this still is an open issue (https://issues.apache.org/jira/browse/HADOOP-8468). VMware created a solution for this problem by introducing Hadoop Virtualization Extensions (HVE) (https://www.vmware.com/files/pdf/Hadoop-Virtualization-Extensions-on-VMware-vSphere-5.pdf). This however is not a general solution that can be used for other distributed data-platforms. We would like to develop a more general solution for this problem by implementing a software layer that can deliver virtualization information (network topology, physical location of VMs, etc.) to distributed data-platforms to help them with task scheduling, data replication and data distribution. As a proof of concept we would like to implement this software layer to improve MongoDB so that it can run on a VMware managed cloud. In order to implement this proof of concept three parts of software need to be created: 1, VMware adapter; 2, API that can deliver platform information to data-platforms; 3, replication and scheduling policies for MongoDB that use the platform information provided by 2. If time allows it we also plan to implement Hadoop extensions to show the wide applicability of the model.

D

Example Sonar Report

1. Diva: VMware Extension

This chapter presents an overview of the project measures. This dashboard shows the most important measures related to project quality, and it provides a good starting point for identifying problems in source code.

1.1. Report Overview

Static Analysis

Lines of code
2,094
8 packages
28 classes
173 methods
0.0% duplicated lines

Comments
34.3%
1,091 comment lines

Complexity
2.3
14.0 /class
391 decision points

Dynamic Analysis

Code Coverage
86.2%
198 tests

Test Success
100.0%
0 failures
0 errors

Coding Rules Violations

Rules Compliance
99.5%

Violations
9

1.2. Violations Analysis

Most violated rules	
Methods should not have too many parameters	1
Constant names should comply with a naming convention	1
Methods should not be too complex	1
Magic Number	1
Unused Imports	5

Most violated files	
PhysicalTopologyProvider	1
VMStartTask	1
VirtualResourceProvider	1
VmBootStatus	1
VSphere	1

Most complex files	
VSphere	114
DefaultVMware	57
VCloud	50
VirtualResourceProvider	36
PhysicalTopologyProvider	18

Most duplicated files
No duplications

E

Nebu RESTful API Specification

E.1. Nebu Core API

Detailed information about each API call is provided below. The `PUT` And `DELETE` calls that are shown in Figure 5.2 exist Only for experimental purposes and are not used in production. No further detail Will be provided for these calls.

`/app` Allows users to create new applications and retrieve the list of existing applications. When creating a new application, users should specify the placement policy that should be used.

`/app/{uuid}` Allows users to retrieve descriptions of existing applications and update their placement policies.

`/app/{uuid}/vmtemplates` and `/app/{uuid}/vmtemplates/{templateid}` provide the same functionality for VM templates as `/app` and `/app{uuid}` respectively provide for applications. Nebu uses VM Templates to specify the type of VM the user wants to use. This information includes some general information like resource load, and some VMM specific information such as which VM image to use and where to find it.

`/app/{uuid}/deployment` and `/app/{uuid}/deployment/{templateid}` provide the same functionality for deployment specifications as `/app` and `/app{uuid}` respectively provide for applications.

`/app/{uuid}/deployment/{deploymentid}/start` starts the deployment with id `deploymentid` for the application with id `uuid`. This includes starting the virtual machines, placing them on the correct physical hosts, connecting them to the correct storage devices.

`/app/{uuid}/virt` gets a list of virtual machines that belong to the application with id `uuid`.

`/app/{uuid}/virt/{vmid}` gets a description of the virtual machine with ID `vmid`.

`/app/{uuid}/phys` gets physical topology information for the application with ID `uuid`. Rather than the complete physical topology of the cloud, this call returns that part of the physical topology where this application is allowed to place virtual machines. Where an application can place VMs is determined by the VM templates that are provided.

E.2. Nebu VMM Extensions API

This API is kept very generic to provide high compatibility. Every basic virtual-machine management system should be able to implement this API. More detail about each API call is provided below.

`/virt` Retrieves a list of all virtual machines visible to the virtual machine manager.

`/virt/{uuid}` Allows Nebu to get information about a specific virtual machine and remove the virtual machine when it is no longer needed.

`/phys` Exposes the physical topology of the cloud. This physical topology should be expressed in Nebu's physical topology format. Nebu uses a generic representation for physical topology for compatibility reasons. This representation is discussed in Section 5.5.

`/phys/{uuid}/createVM` Allows the user to create a new virtual machine. The `uuid` is the unique identifier of the physical host where the virtual machine should be placed. The call has additional parameters that specify the VM template to use, network storage the virtual machine should use, and the hostname the new machine should have. This call returns an SHA hash that allows for status monitoring. See `/status/{sha}`.

`/vmtemplates/{uuid}` Updates an existing VM template, or creates a new template. The VM template consists of the VMM specific information that the user provided when creating a VM template through the Nebu core API. This template contains whatever information the VMM needs to successfully deploy a new virtual machine. The format of this information can be specified by the virtual machine manager that is used.

`/vmtemplates/{uuid}/phys` retrieves a subset of the physical topology that contains all physical hosts and network storage devices that are available to virtual machines of the specified template.

`/status/{sha}` Allows Nebu core to monitor the status of new virtual machines that are being deployed.