AGH University of Science and Technology
in Kraków, Poland

Faculty of Computer Science, Electronics and Telecommunications

Department of Computer Science

Massively Self-Scalable Platform for Data Farming

Dariusz Król

Doctoral dissertation
Computer Science

Supervisor: Prof. Dr. Jacek Kitowski

Kraków, September 2013

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

Katedra Informatyki

Masywnie samoskalowalna platforma wspierająca eksperymenty typu "data farming"

Dariusz Król

Rozprawa doktorska

Promotor: prof. dr hab. inż. Jacek Kitowski

Kraków, wrzesień 2013

Acknowledgements

I would like to thank my supervisor Professor Jacek Kitowski for his invaluable help and research guidance. He offered his full support, valuable technical input and healthy criticism.
Special thanks are due to my colleagues from the Computer Systems Group at the University of Science and Technology AGH for helpful advice, discussions and technical input: Łukasz Dutka, Bartek Kryza, Renata Słota, Michał Wrzeszcz and Włodzimierz Funika.
I am very grateful to my wife Marzena and my parents for their continuous support, encouragement and patience during my research. This dissertation would not be possible without their help.
I am also indebted to the Academic Computer Centre Cyfronet AGH for providing the infrastructure necessary to conduct experimental evaluation of the presented concepts. I would like to thank Łukasz Flis, Marek Magryś and Patryk Lasoń for their help in preparing the testing environment.
This research was partially supported by the European Defence Agency project A-0938-RT-GC "EUSAS", the Polish National Science Centre grant no. 2012/05/N/ST6/03461 and the European Regional Development Fund program no. POIG.02.03.00-00-096/10 as part of the PL-Grid Plus project.

Abstract

In many disciplines of modern science new discoveries can be made by sifting through large quantities of data. Big data is generated, collected and analyzed in both physical and virtual experiments simulating various phenomena with computerized simulations. Recent technological advances have led to significant improvements in computationally-heavy disciplines by providing IT infrastructures capable of executing large-scale simulations in a short amount of time. To effectively exploit these opportunities new scientific methodologies such as Data Farming have emerged. Efficient experimentation using these new tools requires dedicated software, providing (among others) self-scalability - a highly desirable feature which nevertheless remains difficult to implement.
In this dissertation the author introduces two concepts which can be utilized to develop self-scalable platforms, namely self-scalable services and scaling rules. Self-scalable services are an extension of the Service Oriented Architecture which extends the traditional concept of a service to include self-scalability in a standard manner. Scaling rules are a machine-processable notation for defining conditions along with metrics and actions concerning scalability management.
To demonstrate the proposed concepts, a massively self-scalable platform for Data Farming applications is proposed. The functional requirements of this platform are evaluated within the context of multi-agent simulation, which aims to enhance training of security forces in the EDA EUSAS project. The non-functional requirements are evaluated via a set of synthetic tests involving massive scalability and self-scalability under different resource configurations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the motivation for the presented work, along with areas which are especially important for the dissertation, namely the data farming methodology, self-scalability, and heterogeneous computational infrastructures. The author descriptively defines two main problems facing modern data farming software, which will be further investigated in this dissertation. Finally, the main thesis is formulated, along with research objectives and research methodology adopted by the dissertation.

1.1 Motivation

Many disciplines of modern science rely on gathering and analyzing large amounts of data. These disciplines are often collectively referred to as data-oriented (or data-intensive) science. The situation is a consequence of a major paradigm shift which began several years ago. Historically, three other scientific paradigms can be distinguished:

1. *empirical*, which develops science solely by experimentation and observation,
2. *theoretical*, which introduced mathematical formulae to describe the observed phenomena,
3. *computational*, which utilized computers to simulate phenomena too complicated to represent analytically.

Regardless of the adopted paradigm, the key scientific method can be defined as *experimentation*. An experiment is "*a series of tests conducted in a systematic manner to increase the understanding of an existing process or to explore a new product or process*" [15]. An experiment is an essential source of information about processes obtained through observation as well as a method of evaluating theories. The introduction of computers has resulted in an evolution of experimentation. Today, most scientific experiments are assisted by computers, if only to collect and

store all data produced by the experiment. In addition, more and more experiments are conducted in virtual reality – these are referred to as *virtual experiments*.

Large-scale experimentation supported by computers, e.g. in physics, can generate petabytes of data per day. In such experiments data produced by various sensors is stored for further analysis in order to increase our understanding of natural processes. Sometimes it is impossible to collect enough data about a given physical phenomenon in a single experiment and many experiment runs are required. An example is provided by the Large Hadron Collider (LHC), where numerous particle collisions are analyzed to answer fundamental questions regarding the nature of matter.

Some physical experiments – for instance those which require expensive equipment such as airplane engines or military vehicles – are too expensive to repeat enough times to amass meaningful data. In such cases physical experiments are preceded by their virtual counterparts which aim to minimize the number of unknowns and require a fraction of the physical experiment's cost. In addition, virtual experiments are utilized when physical experiments become impossible to perform, e.g. to study crowd behavior during natural disasters.

Utilization of digital devices to support physical and virtual experiments has led to a virtual data "flood". The amount of data generated worldwide is greater than the combined processing capacity of all the world's computers [16]. A special term – "big data" [17] – was coined to describe vast amounts of data which are difficult to process using commodity software within a tolerable period of time. As new scientific findings emerge through analysis of data gathered from various scientific experiments, this new scientific paradigm, sometimes called "The Fourth Paradigm of Science" [18], relates to *data exploration*. Data mining methods [19] become a crucial tool for analysis and knowledge extraction from collected data. Equally important is the systematic process of generating and analyzing data with virtual experiments, based on the methodology known as data farming [20]. This process will be described in detail in the following sections.

An important tool utilized in virtual experiments is *computer simulation* – or *simulation* for short – capable of *representing a portion of the real world with a computer program to study natural phenomena in virtual reality only*. Each simulation involves:

- certain input parameters,
- a model of the simulated entities,
- output, which is also referred to as Measures of Effectiveness (MoE).

MoEs are a set of measurable attributes that describe a meaningful aspect of the simulation. In addition, a simulation can produce text or binary data, e.g. logs of each performed simulation step.

A crucial requirement for performing data-intensive virtual experiments is the usage of high-performance and high-throughput computer infrastructures, particularly when a large number of complicated simulations need to be executed simultaneously, producing results which are aggregated afterwards. Such virtual experiments often demand more computational power than a single computing center is able to provide, requiring integration of organizationally distributed resources. Moreover, new types of computational infrastructures have emerged in recent years, e.g. Cloud environments, offering features distinct from traditional computing clusters or Grids. These aspects need to be considered when planning data-intensive experiments.

It becomes clear that new software for supporting data-intensive virtual experiments is required. Existing software does not enable scientists to take full advantage of all available computational resources, even though they may have access to more resources than ever. This new software should facilitate all phases of conducting data-intensive virtual experiments. In particular, it should virtualize access to computational and storage resources. Besides fulfilling functional requirements, such software should be massively scalable to cope with large-scale virtual experiments.

1.2 Data Farming

Recent technological advances have led to significant improvements in computer simulations, reducing the time required to run a simulation and enabling refinement of simulation models with regard to their complexity. Complicated natural phenomena, e.g. climate changes, can now be simulated in a reasonable amount of time. Besides accelerating simulations, modern high-performance computing infrastructures are capable of processing much more data in a given interval than ever before. New data mining and statistical data analysis tools are also emerging at a rapid pace. As a result, many complex phenomena – such as flood scenarios – can finally be modeled in real time.

Based on this technological progress, new scientific methodologies centered around data-intensive computation and analysis have emerged. Data farming [21, 22] is an example of such a methodology where the main objective is to obtain better understanding of the analyzed phenomena by examination of entire landscapes of potential outcomes – not just selected cases – through data-oriented virtual experiments. Data farming utilizes high-performance and high-throughput computing to generate large amounts of data via computer simulations. These results are subsequently analyzed to obtain new insight into various phenomena. Hence, data farming can be considered to represent "the Fourth Paradigm of Science".

The data farming methodology is well suited for studying complicated multi-parameter scenarios which cannot be efficiently solved with analytical methods, e.g. involving fuzzy variables such as leadership or trust. Initial applications of data

farming concerned verification and enhancement of existing procedures and analytic culture at the Department of Defense [23]. One data farming application developed within this project aimed to facilitate the choice between maneuver and attrition in combat scenarios. The simulation model involved Red forces as defenders and Blue forces as attackers. MoEs of this simulation included the number of eliminated Blue entities and whether or not Blue forces where prevented from penetrating the area defended by Red forces. Input parameters included the firing range of Red forces, their accuracy and the attack strategy of the Blue forces (heading straight for the objective or attempting to outmaneuver the enemy). By running multiple simulations, it became possible to determine that maneuvering is superior when Red forces possess long range and high accuracy, while heading straight for the objective is advisable in all other cases. Moreover, based on this information, analysts decided to enhance the simulation model with aggression and sensor range parameters for the Red forces, and then run the next batch of simulations. Based on these new results they deduced that Red forces should be more aggressive in order to increase their effectiveness.

Data farming refers to the process of conducting virtual experiments, also referred to as *data farming experiments*, which follows the methodological principles depicted in Fig. 1.1. In order to be considered a *data farming experiment*, the virtual experiment should consist of the following steps:

1. *Experiment objective definition* is an initial step which involves stating questions and objectives which should be answered and achieved by the experiment. In addition, a stop condition is formulated as the data farming process can be iterated many times before stopping.

2. *Simulation scenario building* concerns providing a simulation capable of generating the necessary data to answer the questions stated at the beginning of the experiment. Hence, it is necessary to select or develop a simulation model with necessary input parameters and meaningful MoEs.

3. *Input space specification* results in a set of input vectors, each of which represents a single simulation case. As the input space can be extremely large, Design of Experiment (DoE) methods [24] are often employed to reduce the number of input vectors. These methods may include two-level and fractional factorial design.

4. *Simulation execution* involves execution of simulations with input vectors generated in the previous step. Each data farming experiment comprises multiple simulations, often run in parallel using High Throughput Computing (HTC). Depending on the input space this step can require organizationally distributed

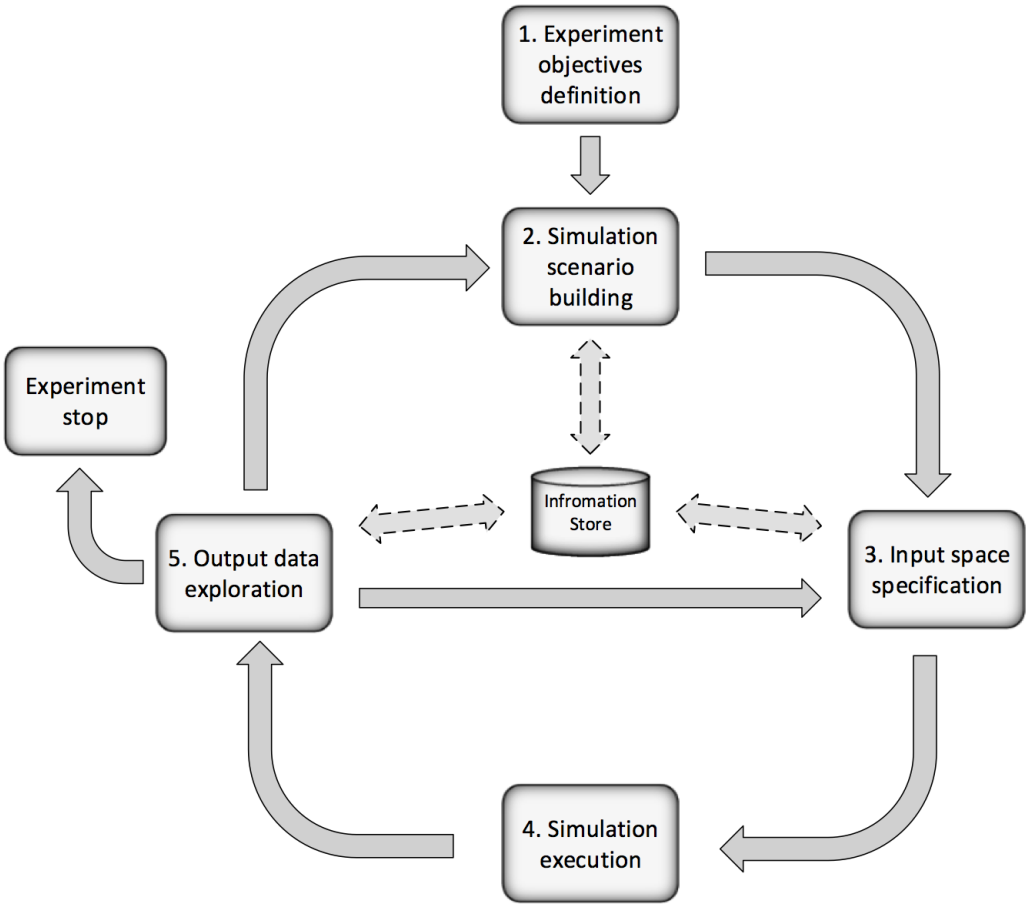

Figure 1.1: The process of a data farming experiment.

resources working together to provide the necessary computing power. Results from all simulations are aggregated for further analysis.

5. *Output data exploration* is where knowledge is extracted and new insights obtained. Should a simulation scenario require adjustments, step 2 may be repeated as needed. If additional areas of the parameter space need to be explored, step 3 is repeated. Otherwise the stop condition is considered fulfilled and the experiment concludes.

The "Information Store" concept refers to a knowledge base, which collects all relevant information throughout the experiment. This information can be utilized at subsequent steps of the process to increase efficiency. One exception is the "Simulation execution" step, where data is generated for further use. On the other hand, the collected information is necessary to meet the objectives of the experiment.

Data farming combines several existing concepts and techniques into a coherent process facilitating data-oriented virtual experiments:

- parameter study and DoE methods to specify the experiment input space,
- data exploration with data mining and other statistical methods to extract knowledge from multiple simulation results,
- HTC to run multiple simulations in parallel to minimize the time required to carry out experimentation.

1.3 Self-Scalable Software

The second area of computer science seen as important to this dissertation concerns automatic software management. Here, a common use case involves a web service which starts as a small project for a limited group of clients but becomes very popular over time. At first, this service can be monitored and maintained manually by a single administrator; however as the number of clients increases, a need for additional resources emerges and manual maintenance becomes difficult and ineffective. It is therefore crucial for the service to be able to manage itself. A special type of management, called *self-scalability*, is especially important when dealing with unpredictable and dynamic load conditions.

Scalability can be defined as *the ability of a computer program to cope with increased workload*. In this sense an ideal scalable computer program should retain performance when confronted with a heavier load but also provided with an increased quantity of resources. In the context of High-Performance Computing (HPC), performance relates to FLoating-point Operations Per Second (FLOPS) when executing a single large task. On the other hand, the performance of HTC systems is measured by the number of independent tasks executed per second. This dissertation focuses on HTC systems due to the nature of data farming experiments.

As with algorithms [25], we can define three levels of scalability:

- *Linear scalability*, where additional resources of a given type always increase application performance by the same amount.
- *Sub-linear scalability*, where adding resources has diminishing influence on application capacity, e.g. due to synchronization overhead.
- *Super-linear scalability*, where additional resources of a given type contribute more than the same amount of additional capacity to the application.

In practice, due to multiple sources of overhead (e.g synchronization and communication), a computer program is considered scalable when additional resources contribute to its performance in a similar way, i.e. the difference between performance contributions provided by each batch of resources is not significant.

In most cases an application is scaled manually: upon discovering a change in workload patterns the operator allocates new resources and reconfigures the application. In contrast, a self-scalable application scales itself without any external interaction. Once configured, it can adjust itself to different workload patterns dynamically.

Advantages of self-scalability relate to the effort required to maintain applications in a running state. Traditional applications often require constant monitoring and possible reconfiguration when the workload pattern changes, e.g. when the number of clients increases or resource usage efficiency drops as a result of competition with other applications. In many situations a human administrator needs to be present at all times simply to ensure that a critical application continues to operate. On the other hand, self-scalable applications should perform administrative actions automatically. Moreover, by using monitoring data, self-scalable applications can be more efficient than their manually operated counterparts due to faster reaction time. This is especially important in dynamically changing environments when the workload pattern cannot be determined or predicted beforehand. Building a self-scalable application can be a challenging task. Such functionality is typically implemented in a separate module, often referred to as the management module, responsible for analyzing application load based on monitoring data and executing scaling actions, e.g. starting a new instance of the application on a different server. The management module typically implements the following four features:

- *online monitoring*, i.e. collecting online data about current application workload,
- *detection of events*, which should trigger the above mentioned scaling procedure,
- *scaling procedure execution*, which involves acquisition of additional resources by the application,
- *resource discovery*, which encompasses identification of resources that can be used during the scaling procedure.

Besides implementing these features, self-scalable applications require knowledge about events that should trigger the scaling procedure. This knowledge can assume the form of rules which define conditions under which the management module should perform certain actions. Such rules are often gathered by observing the

application in real-life scenarios and may be difficult to generate automatically. Thus, the decision to enhance an existing application with self-scalability features is not an obvious one.

Self-scalability belongs to a popular set of features often referred to as self-*, which denotes features related to application autonomy. The set also encompasses the following capabilities:

- *self-healing*, which is the ability of a system to automatically recover from a failure,
- *self-organization*, which is the ability of a system to dynamically adjust its logical or physical organization to new requirements at runtime,
- *self-adaptation*, which is the ability of a system to adapt itself to a changing environment in an automatic manner,
- *self-protection*, which reflects the need for proactive authentication and protection from attacks.

These properties are used to describe systems which should provide a high level of automatic behavior and can be considered self-aware. Such systems are the subject of Autonomic Computing initiative [26] research. The initiative intends to provide mechanisms and tools for developing intelligent, self-managed computing systems, where administrators' interference is reduced to a minimum. The inspiration for this research is the human autonomic nervous system, which controls key functions without any outside involvement. One way to design an autonomic system is to extend an existing portion, responsible for functional requirements, with a component which takes care of non-functional requirements, e.g. availability. A possible design of such a component is depicted in Fig. 1.2. The basis of every possible action that can be performed by the component is knowledge about the system as a whole. By performing, in a loop, the steps depicted in Fig. 1.2, namely monitoring, analyzing, planning and executing, the component can implement self-management of resources.

1.4 Heterogeneous Computational Infrastructures

Modern scientific research requires efficient resource sharing between various institutions and initiatives. This requirement is dictated by two key factors: scientific research is often performed by teams from multiple geographically distributed institutions and moreover, computational power and storage capacity required to perform scientific experiments often exceed the capacity of a single data center. Over the

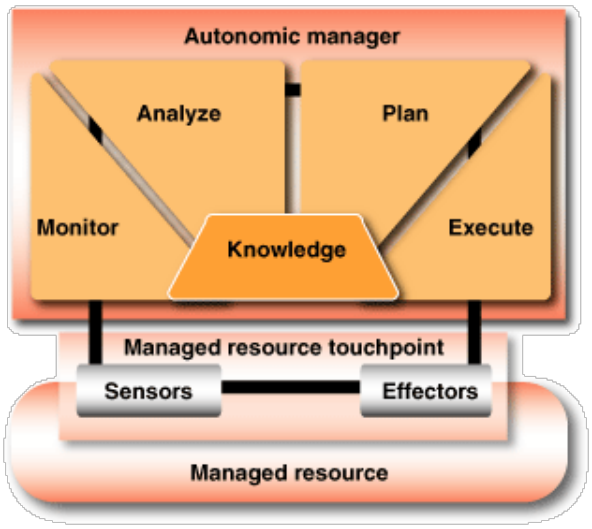

Figure 1.2: An autonomic computing manager [1]

recent years, great effort has been expended to design and implement distributed environments which would enable users to access shared resources in a uniform manner. Historically, each scientific facility set up its own data center which varied in size depending on the perceived user needs. A single data center would host computing and storage resources along with other more specialized instruments, e.g. chemistry or material science labs. All resources were connected with a network and thus accessible remotely. Depending on the scope of research, the yearly level of utilization of a data center's resources was between 30% to 50% [27]. On the other hand, when a scientific facility did not possess sufficient funds to build its own data center, its members had to apply for help to a nearby affiliated facility which possessed such a center. This was generally a cumbersome and time-consuming process.

Another issue related to resource sharing began to emerge as the amount of resources at each data center increased (along with the number of users who intended to utilize these resources). The issue relates to the effort required to maintain a fair-share policy (in terms of application execution) on the one hand, and to maximize resource utilization on the other. A data center is a multi-tenant pool of resources which should be accessed in a uniform and intuitive way by its users. Back when computing and storage resources were severely limited, this issue was mitigated by the small number of users and reservation-based scheduling. This was a quasi-optimal situation, since a single user would often use most of the available resources. Currently we are facing the opposite scenario, i.e. a data center can handle hundreds or thousands of users simultaneously while a single user needs only a small fraction of the available resources. Thus, a different approach is needed to facilitate efficient scheduling and resource management. Moreover, it would be desirable to decrease

the amount of administrative effort by utilizing dedicated applications to monitor and perform any necessary actions automatically. In fact, many researchers predict that future data centers will delegate routine administrative tasks and actions to custom software since the amount of resources will become unmanageable for a human administrator [28].

The scientific community, along with various commercial providers, has invested a lot of work in resolving the problem of efficient and transparent resource sharing across administrative boundaries. Although the stated problem seemed simple, its resolution under real-world conditions turned out to be anything but trivial. Two of the most successful solutions in this regard are computing Grids and Clouds. Both ideas grew out of the desire to make computational power and storage capacity accessible in a similar way to other basic utilities such as electricity or telephone links.

Grid computing [29] intended to implement this idea by providing an additional software layer (middleware) between users and resources, responsible for resource access management and application scheduling in a organizationally distributed environment. Grid computing envisioned a coherent distributed environment with several points of access to the underlying resources, which would be shared across multiple institutions. This idea underpinned multiple scientific research projects. Many software frameworks and toolkits were created and a significant amount of computational and storage resources are currently shared in Grid environments.

On the other hand, Cloud computing [30] is a more business-oriented approach. Originally, Cloud computing was invented to increase utilization of IT resources at large companies such as Amazon or Microsoft. Since corporate data centers were built to handle peak load scenarios, which rarely materialized in practice, their resources were often severely underutilized. Thus, corporations began to rent out computational resources to third parties via a pay-per-use model. An important part of this solution was a set of virtualization techniques for computational and storage resources, which enabled Cloud providers to maintain effective separation between different clients executing code on a single machine. Another important goal was to make the rent process as simple and intuitive as possible, which, in practice, meant that the client could start a new virtual machine with just a few clicks. As a result, Clouds minimize investment risks by reducing the initial infrastructure costs, which leads to more applications being exposed as online services.

1.5 Problem Description

The efficiency of doing scientific research with the data farming methodology is highly dependent on the available software. This is due to the need to manage high-performance and high-throughput computational infrastructures which run multiple

simulations in parallel, and to aggregate their results. Thus, in order to increase the efficiency of virtual experiments based on the data farming methodology, we first have to provide software which supports this methodology. By investigating existing software and the data farming process itself we have identified three main problems to be addressed by this dissertation:

- lack of scalability,
- poor utilization of computing and storage resources,
- poor integration with different computational infrastructures.

An example of a common situation involving modern software for running multiple simulations in parallel is depicted in Fig. 1.3. By following the "master-worker" design pattern, components of such software are divided into two groups: managers and workers. Managers, which constitute the "master" part of the software stack, are responsible for preparing the input parameter space, assign its elements to workers and collect any output. "Workers" reside on computational resources and perform actual simulations using the supplied input values.

When considering data farming experiments with tens of thousands of simulations, the platform should be able to run not only a large number of workers but also multiple managers. The actual quantity of each component type depends on the simulation in question:

- simulations which take a short time to complete typically produce heavy communication overhead and demand more throughput from managers,
- lengthy simulations decrease the amount of required communication per unit of time – as a result the number of managers can be much lower.

The number of workers is typically much higher than the number of managers (although the ratio can vary dynamically).

Self-adjustment of software to different workload patterns is a descriptive definition of self-scalability. It is especially desirable in highly dynamic environments such as Clouds, which can provide computing resources on a large scale. Achieving high throughput and minimizing the cost of running independent tasks in Clouds is the focus of methodologies such as task farming [31]. However, most of the existing software packages for running jobs on computational infrastructures only address worker scalability, i.e. adjust the number of workers to suit the experiment's demands in order to achieve the highest throughput. The greater the influence of manager throughput on experiment efficiency, the more important manager scalability becomes. In many cases a predefined pool of generic computational resources

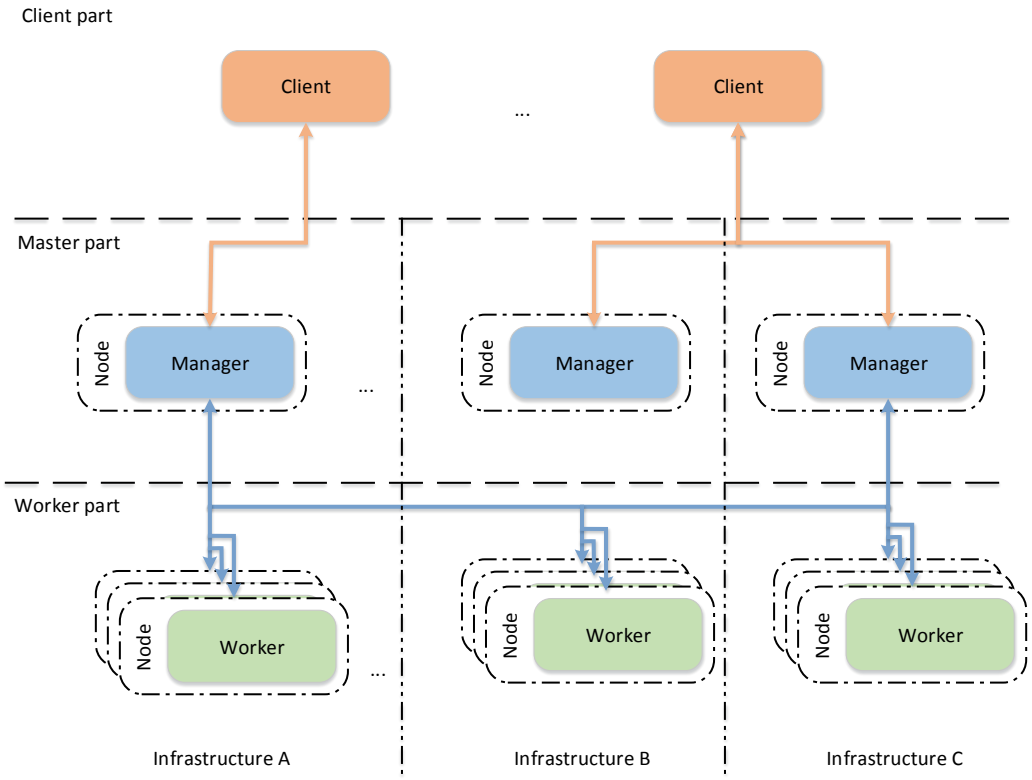

Figure 1.3: A virtual platform for running experiments.

is used both to run both managers and workers. In such a scenario downward scalability (the ability of a system to decrease the number of running managers in order to run more workers) becomes very important.

Although the infrastructure for a data farming experiment can be configured to handle peak throughput requirements, such an approach can lead to low utilization of computing resources, especially when running short simulations which require more communication between workers and managers to exchange information about subsequent simulations. A similar situation can occur when a business – e.g. Amazon Inc. [32] – invests in a data center to run a popular service. As the service only operates under peak load conditions for several days per year, the data center remains underutilized throughout the rest of the year. Hence, it might be desirable to deallocate some managers to free resources for additional workers, increasing the overall resource utilization level and reducing the time required to perform data farming experiments.

The ratio between managers and workers should therefore be dynamically adjusted to match the changing workload. This adjustment should be performed in an

automatic manner, though it should take into account expert knowledge expressed in the form of scaling rules, i.e. conditions upon which managers or workers should be scaled upward or downward. Scaling rules should reflect measurable parameters, e.g. service response time, CPU, memory or measurement aggregation, along with interpretation methods such as capturing the average measurement within a given time frame or discovering trends in online workload. For each specific situation scaling rules should be set in advance by an expert who has in-depth knowledge about the properties of a given simulation. This batch of scaling rules can be treated either as a final set or as an initial point. In the latter case, the system should exploit information gathered at runtime to adjust scaling rules appropriately.

Last but not least, executing a large number of simulations may exceed the capacity of a single data center. The platform should be capable of scheduling simulation execution between organizationally distributed infrastructures, as depicted in Fig. 1.3. By such infrastructures we mean distributed environments which provide computational and storage resources with a known interface, e.g. a Grid scheduling system, a Cloud environment available through a vendor-specific SDK, or even an institutional cluster accessible via Secure Shell.

1.6 Thesis Statement and Research Objectives

As described in previous subsections, there is a strong need for enhancing the data farming methodology with an efficient software platform that will support each phase of the data farming process. Based on this requirement the following thesis will be investigated in this dissertation:

Platforms for data farming processes require a heterogeneous computational infrastructure and support for self-scaling in order to provide efficient and cost-effective performance.

Therefore, the main goals of the proposed thesis are as follows:

- to design and implement a massively self-scalable virtual platform which supports each phase of the data farming process and utilizes a heterogeneous computational infrastructure,
- to propose a set of scaling rules, which take into account time- and cost-related parameters and ensure a high level of performance with regard to user requirements concerning costs.

The research methodology for validating the proposed thesis includes:

- development of a virtual platform,
- preparing a set of synthetic tests for validating the platform's scalability,
- validating the self-scalability feature of the platform with synthetic tests and sample data farming experiments which generate different workload patterns and hence require different scaling actions,
- investigating other areas where the platform can be utilized.

1.7 Note on Participation in European Research Projects

The author of this dissertation is a member of the Knowledge in Grids Team at the Department of Computer Science, AGH University of Science and Technology and has participated in several EU-funded research projects as an employee of the Academic Computer Centre CYFRONET AGH. This dissertation has been influenced by experience gained in the course of the above mentioned work.

Participation as a scientific developer in the EU-IST ViroLab [33, 34] and EU-IST GREDIA [35] projects provided insight into development of Grid collaborative platforms for e-Science [36], user interfaces [37, 38] and infrastructures [39].

Development of a semantic-oriented monitoring tool within the POIG IT-SOA [40] project has increased the author's knowledge about (SOA) [41], QoS-oriented (Quality of Service) monitoring systems [42, 43, 44] and semi-automatic management of distributed applications [45, 46].

During the PL-Grid [47, 48] and PLGrid Plus [49] projects the author was responsible for highly scalable, semantic-based data management systems working in both Grid and Cloud environments [50, 51, 52], obtaining insight into scalability problems facing large-scale applications in heterogeneous computational infrastructures [53, 54, 55, 56, 57].

Participation in the European Defense Agency (EDA) EUSAS [14] project as a key developer of the data farming platform [58] enabled investigation of the data farming methodology and issues related to existing software which supports this methodology [59].

1.8 Thesis Contribution

The work performed within this thesis contributes to three areas of computer science: scalability management, software engineering and data farming, with the following

elements:

- The author proposes the concept of *scaling rules* – a formal way of expressing scaling management knowledge. For a given platform scaling rules describe how the system should rescale itself in response to various conditions. Such rules can be predefined by domain experts and then utilized automatically by computer systems.
- In order to address the scalability requirements of modern distributed software platforms, an extension of SOA, called *self-scalable services*, is proposed, acknowledging the scalability property as a first-class citizen of software architectures. A self-scalable service extends the meaning of a software modularization unit with built-in self-scalability.
- The presented concepts were exploited during development of a massively self-scalable virtual platform for data farming called *Scalarm*, which is a complete solution for performing large-scale data farming experiments using a heterogeneous computational infrastructure with minimal administrative effort.
- To verify Scalarm's scalability and functionality a number of experiments were conducted using both synthetic and real-life scenarios.

1.9 Thesis Structure

The thesis is organized as follows: Chapter 2 provides an analysis of existing data farming systems, self-scalable solutions, systems for data storage and computational environments. In Chapter 3 the author introduces the virtual platform for data farming, starting with user requirements through platform design and architecture definition. As part of of this chapter the concept of self-scalable services is presented. Chapter 4 discusses the problem of scalability in the context of a virtual platform for data farming. Moreover, the concept of scaling rules is introduced. In the following chapter 5 an implementation is described for both self-scalable services and scaling rules. A reference implementation of both concepts is provided in the form of a massively self-scalable platform for data farming experiments called Scalarm. Additionally, a thorough description of all platform components and representative use cases which explicitly depend on the platform's scalability features, is provided. Chapter 6 contains a detailed description of a complete Scalarm experimental evaluation. This evaluation is divided into two parts. The first part concerns the scalability feature of the platform, while the second part is related to the self-scalability aspect. In Chapter 7 a real-life application of the Scalarm platform is described in the context of enhancing training of security forces. Finally, a summary with possible directions for future work is presented in Chapter 8.

Chapter 2

Background Survey

This chapter describes various ongoing work in areas related to this thesis. Since the data farming approach includes task scheduling, simulation management and data storage, it is important to explore work related to these topics. In addition, we provide an overview of self-scalable systems and computational infrastructures which focus on heterogeneity and scalability issues.

2.1 Data Farming Systems

We begin our background survey by identifying relevant research from other projects and activities which focus on building systems that either directly support Data Farming or deal with a subset of Data Farming phases, e.g. simulation scheduling. In particular, we explore scalability, support for heterogeneous computational infrastructures, data analysis methods and the ability to conduct Data Farming experiments in an exploratory way.

2.1.1 OldMcData

Although Data Farming is becoming quite widespread, software which supports this methodology remains limited in scope. One of the most popular examples is OldMcData - the Data Farmer (OMD) [60], a small-scale system that can execute multiple simulations on a standalone computer or in a distributed computational network. It was developed at the SEED Center for Data Farming [61] and integrated with external tools to support preparation and execution of data farming experiments.

OMD utilizes an application called Xstudy [62] to set up data farming experiments. Xstudy uses an XML file, called study.xml, to specify information about the simulation model, input parameters of the experiment, the type of algorithm which should be used to generate the actual set of input parameter values, and other administrative data such as the user's contact details. As a text file, study.xml can be created and edited using any text editor, however Xstudy provides a user-friendly graphical frontend to carry out all the preparatory steps and initiate execution. In

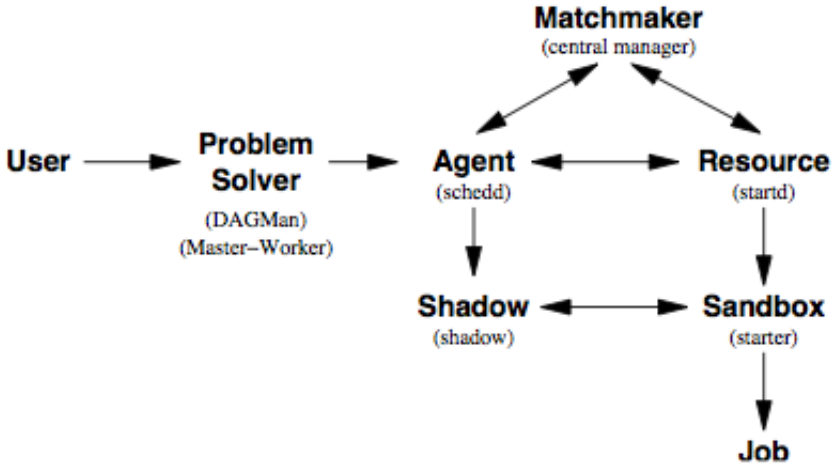

Figure 2.1: Architecture of the Condor distributed scheduler [2].

addition, Xstudy can import a list of comma-separated values (CSV) describing an experiment.

By applying the study.xml file, OMD can generate separate configurations for each simulation design point, which is defined as a vector of input parameter values. It finds a base scenario file which contains a complete configuration for running a simulation but without actual values of input parameters. Since it is an XML file, OMD locates all input parameter elements and substitutes actual parameter values for each design point with the selected Design of Experiment (DoE) algorithm. As a result, a new configuration file is created. Currently OMD supports the following DoE algorithms: full factorial, Cartesian product, values specified in a CSV file and evolutionary programming. Moreover, several parameters can be grouped and assume the same values for selected simulations.

OMD schedules simulations to run either on a standalone computer or on available distributed computational resources using Condor software [63], whose goal is to provide mechanisms and policies that support HTC on large collections of distributed resources. Condor supports Grid and Cloud environments via the Globus toolkit [64]. The most important processes in the Condor system are depicted in Fig. 2.1. The user (in our case, OMD) submits jobs to an agent which is responsible for finding suitable resources. Agents and resources are registered in a matchmaker which can introduce compatible agents and resources. Upon finding a match the shadow component of the agent provides all the required details about a job to the sandbox in order to create a safe execution environment. Once the job is completed its output can be moved from the resource to a designated point.

Unfortunately, no data analysis methods are provided by the OMD. This introduces the need for external tools appropriate to the output format of the simulation. Moreover, running simulations is a batch-like process, i.e. the entire input package is submitted to the scheduler all at once. The user cannot proceed with data analysis until the experiment is finished. There is no information about partial results and the user cannot modify the set of input vectors after submission. Although Condor can be integrated with heterogeneous infrastructures, it lacks self-scaling features, which means that the infrastructure used to run the experiment has to be set up beforehand and cannot change at runtime.

2.1.2 JWARS

The Joint Warfare System (JWARS) [65] is a virtual platform for running a campaign-level model of military operations. It intends to provide a simulation of joint warfare that supports operational planning and execution, force assessment studies, system trade analyses, and concept and doctrine development. It began as a joint military program funded by the Office of the Secretary of Defense to create a simulation and modeling framework for military operations. JWARS was used in a number of projects to help plan various military deployments and develop military doctrines.

JWARS was one of the first attempts to integrate all phases of a military campaign, from planning through execution to analysis. It supports creating an operational plan from doctrines, rules of engagement, and campaigns while incorporating entity locations and movement. Multiple simulations can be run in parallel and their output gathered and analyzed in the context of force assessment studies and statistical research. JWARS provides an event-stepped simulation system that describes the behavior and interaction of military forces across a wide spectrum of scenarios.

JWARS includes three software domains, namely problem, simulation, and platform. All are integrated into a single package which is used to perform studies and analyses. The problem domain models entities which exist during simulations. The simulation domain provides an engine which executes simulations in a stepwise manner in a three-dimensional battlespace. The platform domain incorporates hardware and a Human-Computer Interface (HCI) which assists analysts in getting data into and out of the simulation. The current version of the platform is based on a client-server architecture in which HCI runs on the client side while the simulation logic and management are located on the server side.

The logical structure of the JWARS platform is depicted in Fig. 2.2. JWARS implements the "Observe, Orient, Decide, Act" simulation loop paradigm. The "Ground Truth" database provides a battlespace abstraction and contains all force descriptions, their plans, possible behaviors, and the environment in which they exist. Information about the opposing force is collected using sensors. The process-

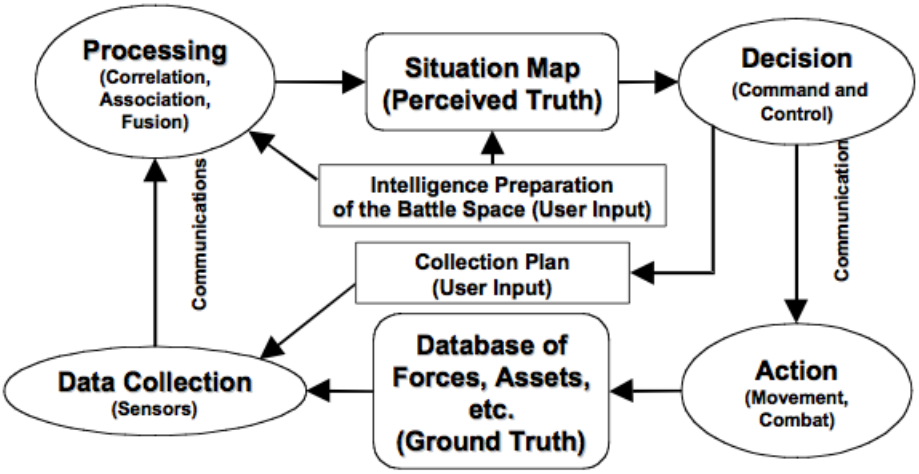

Figure 2.2: Architecture of the JWARS platform [3].

ing node represents the activity necessary to formulate a commander's perception. JWARS utilizes historical perception states and processes incoming information to create a new state. The current state of perception is then used to build a situation map which contains all relevant information (e.g. the position of own troops), necessary to make a decision concerning the next action. Once the data farming experiment concludes analysts can visualize the results using several analysis tools built into JWARS. Alternatively, output data can be exported to a file for further analysis using third-party software.

Due to the potential for involving humans in the simulation loop JWARS focuses on analyzing output from a limited number of simulations rather than on collecting data from millions of simulations. In addition, all runtime information about each running simulation is stored in a central database, which is a single point of failure (SPOF) for the JWARS architecture. As such, the output landscape analysis features are rather limited in complex scenarios (which may involve dozens of parameters). JWARS requires a dedicated cluster, which greatly limits the scale of supported data farming experiments and mitigates the need for self-scaling.

2.1.3 SWAGES

Artificial life (Alife) is another example of a multi-agent simulation environment. Artificial life refers to the concept of studying living systems running in virtual worlds in order to understand the way in which such systems process information. The main idea is to synthesize lifelike behavior from scratch *in silico*. This would allow researchers to investigate non-trivial problems such as the origins of life, self-

assembly, growth and development, evolutionary and ecological dynamics, animal and robot behavior, social organization and cultural evolution. SWAGES is an experimentation platform for distributed agent-based Alife simulations which employs dynamic parallelization and distribution of simulations in a heterogeneous computing environment. SWAGES combines several loosely coupled components to provide a coherent platform for supervising Alife experiments which involve multiple large-scale agent-based simulations. It supports all experimentation phases, namely:

- Setting up experiment sets, i.e. generating configurations for a number of simulations based on initial conditions.
- Scheduling simulations to run on the available computational resources. The scheduling process is based on several priority-based queues to which simulation configurations are submitted for execution on remote hosts.
- Supervising running simulations on remote hosts. SWAGES monitors the progress of running simulations and handles failures of remote hosts by re-running crashed simulations from scratch or from a saved state.
- Gathering output from simulations and exporting the data in a number of formats supported by external tools for further processing.

An important feature of the SWAGES platform is its extendability with third-party components, e.g. result visualization tools or physics engines. Using the so-called open "plug-in architecture", SWAGES allows users to exchange information between its internal and external components via inter-process communication means, e.g. shared memory or network sockets. An existing version of the SWAGES platform uses a general-purpose environment called SimWorld [66], to develop and run agent-based simulations. It supports running simulations in a graphical (interactive) mode as well as in a batch (non-interactive) mode. SimWorld includes an automatic parallelization mechanism which is based on the simulation distribution algorithm. The algorithm can either update all parallel simulations one cycle at a time or independently update simulations after as many cycles as possible (when information from other simulations is needed). The latter mode utilizes spatial "spheres of influence" which describe the range at which one agent affects others.

While SWAGES can use multiple computational resources to perform simulations, its scalability is rather limited. The main reason for this is the need to maintain a connection between the server and each running simulation. Moreover, server-side components of SWAGES lack the clustering feature, even though they can be run on separate hosts. As a result, SWAGES can only be used in experiments which do not entail a large number of simulations. Another limitation is the

graphical user interface, which does not provide any means of analyzing partial results. In order to perform analysis – even a simple one – the user first has to export experiment results and then apply an external tool, e.g. the R statistical language.

2.1.4 DIRAC

Although dedicated data farming tools are rather limited in number, several software packages support selected phases of the data farming process. Among the most important phases is simulation, which needs to run on a high-performance and high-throughput computational infrastructure. As this is a generic problem shared by many areas of computational science, several tools are available. Distributed Infrastructure with Remote Agent Control (DIRAC) [67] is a platform supporting computations with heterogeneous resources including local clusters, Grids and Clouds. It was originally developed to provide a complete solution for using the distributed computing resources of the LHCb experiment [68] at CERN for data production and analysis. However, it remains a generic platform and can interface with non-reliable resources in an efficient way to perform computational jobs.

DIRAC provides an additional abstraction layer between users and various computational resources to allow optimized, transparent and reliable usage. It applies the so-called Workload Management System with Pilot Jobs which increases computational efficiency and reliability. DIRAC is an agent-based architecture where agents are deployed on worker nodes, creating a dynamic overlay network of readily available resources. Agents constitute a representation of the available computing resources. Their goal is to reserve computational power to run actual tasks which are distributed using a custom scheduling model. By applying the Pilot Jobs and Workload Management System concepts DIRAC implements redundancy at the computational task level, i.e. guarantees that tasks will be run and rescheduled in case of failure. In addition, these concepts enable a single system to aggregate various types of computing resources such as computational Grids, Clouds and clusters – all in a manner which is transparent to end users.

DIRAC follows the SOA paradigm; hence it is composed of a number of loosely coupled components, as depicted in Fig. 2.3. These components can be grouped into four categories:

- Resources which provide access to computing and storage infrastructures.
- Services which maintain system state and handle workload and data management tasks. Each service is a passive component which operates by reacting to client requests.
- Agents which run the actual computational tasks on behalf of the user on the available resources. Agents provide a uniform way to deploy, configure, control

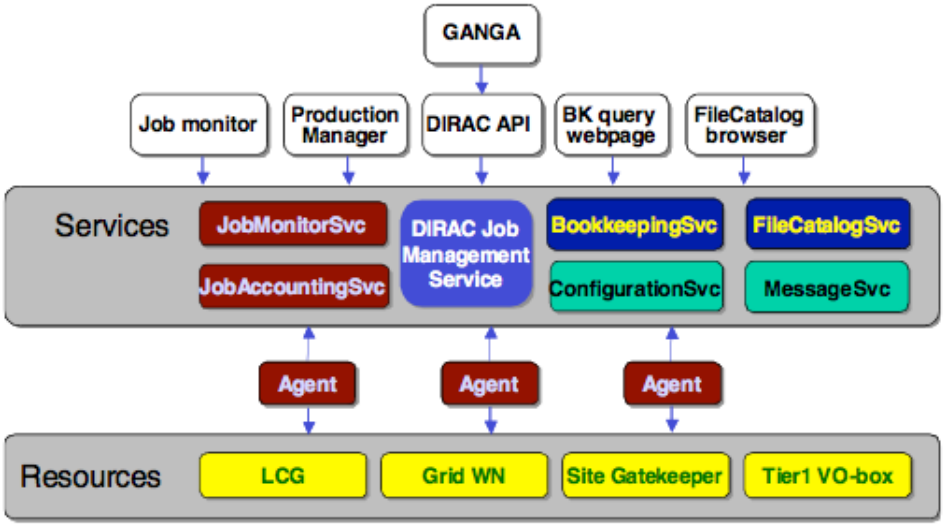

Figure 2.3: Architecture of the DIRAC system [4].

and log their own activity. They can operate in different environments such as Grids, clusters or Clouds.

- Interfaces which are access point to the system from the user's or developer's points of view. The end user uses command-line tools to schedule jobs with DIRAC, while developers can exploit a dedicated Application Programming Interface (API) exposed by the DIRAC platform to implement third-party tools, e.g. Graphical User Interfaces or other abstraction layers on top of DIRAC itself.

While DIRAC provides certain data management features, they are chiefly related to reliable data distribution among computational resources. As a generic tool focused on computations, it does not provide task result analysis extensions. Moreover, it does not have inbuilt DoE methods for sampling the input parameter value space upon which computational jobs should be generated. Thus, it can only be used as a single component of a complete data farming platform rather than a self-contained solution.

2.2 Self-Scalable Systems

Self-scalability is a necessary feature in a massively scalable platform. A platform which comprises a large number of resources should be able to adjust itself to a dynamically changing environment and load.

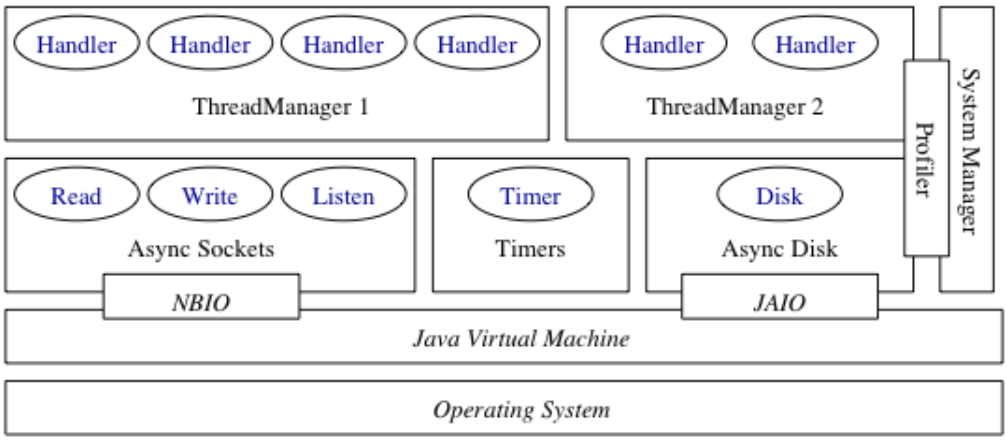

Figure 2.4: Architecture of the reference SEDA implementation – Sandstorm [5].

2.2.1 Staged Event-Driven Architecture

Staged Event-Driven Architecture (SEDA) [69] is a design approach for building highly concurrent server applications. It intends to provide a hybrid infrastructure which utilizes threading and event-driven programming models. Its main goals include enabling applications to be well conditioned to load, preventing computational resources from being overcommitted when demand exceeds capacity. Each application based on SEDA is decomposed into a set of stages, which are similar to states in the event-driven programming model. Stages communicate with each other via messages. Each application's stages can be executed using the threading model. By separating stages, better performance and fault tolerance can be achieved.

The reference implementation of SEDA is called Sandstorm [70]. The platform provides a set of interfaces with which to build applications. Its architecture is depicted in Fig. 2.4. A Sandstorm-based application consists of a set of stages connected by queues. Each stage is implemented as a module with two components: an event handler and a stage wrapper. The handler receives notifications about events that have occurred (e.g. incoming messages) and encapsulates the logic of the application, while the stage wrapper is responsible for creating and managing event queues. The flow of stages is controlled by a thread manager which allocates thread and schedules event handlers for execution. An important aspect of the Sandstorm platform is built-in support for customization through replaceable resource scheduling policies. As such, it is relatively easy to replace the basic thread manager with a more sophisticated implementation. In addition, Sandstorm provides services such as timers and profilers, which can support application development and testing.

Although SEDA is an interesting approach to building scalable software, it is oriented towards fine-grained (as opposed to coarse-grained) concurrency, i.e. executing

modular applications in parallel using message passing. Moreover, its reference implementation is rather limited and has not been completed as of yet. Nevertheless, several open-source and commercial systems, e.g. SwiftMQ [71] or OceanStore [72], are based on SEDA principles such as non-blocking I/O or event-driven programming.

2.2.2 GigaSpaces eXtreme Application Platform

Besides academic research, self-scalable platforms are very important in the commercial world. System malfunctions resulting from excessive client load often incur significant costs – up to millions of dollars per hour of downtime [73]. Thus, it is essential to maintain system stability at all times. There are several commercial platforms which support enterprise virtualization and application scalability. One such platform is called the eXtreme Application Platform (XAP).

XAP intends to provide end-to-end scalability of applications and data under extreme latency and load requirements. It is designed to meet the mission-critical needs of a wide range of businesses with the following features:

- online monitoring,
- advanced management capabilities,
- automation of operations,
- supporting private, public, and hybrid Cloud environments,
- integration with popular programming frameworks,
- interoperability among programming languages, environments, and APIs.

XAP departs from common tier-based applications due to their perceived disadvantages such as management overhead or latency of business transactions which span all tiers. Instead, it proposes a different approach based on the notion of separating processing units which represent self-contained components. Each such component includes processing, data management and messaging. At the core of each processing unit is a scalable, high-performance, reliable in-memory data grid (IMDG). IMDGs support multiple APIs for accessing stored data with a clustered in-memory message bus which supports update subscriptions and cluster-wide code execution. The latter feature enables IMDG to function as a scalable processing environment with shared memory between nodes and built-in support for the map-reduce pattern.

The architecture of XAP is depicted in Fig. 2.5. The Open Interfacing Layer enables uniform access to resources for applications written in various programming

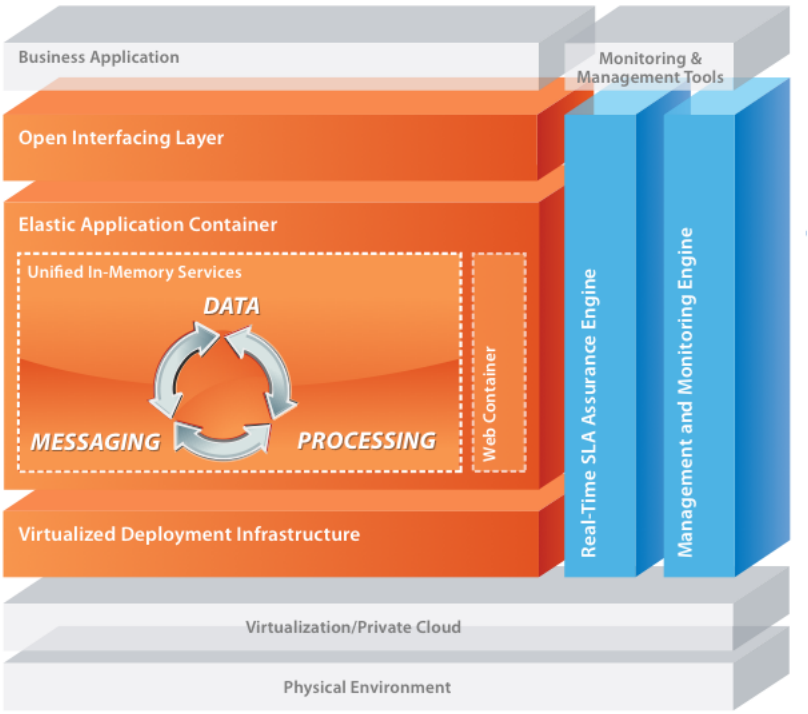

Figure 2.5: Tier-based architecture of a GigaSpaces XAP processing unit [6].

languages and technologies. The Elastic Applications Container is an implementation of the processing unit concept with support for self-scaling based on defined rules. The Virtualized Deployment Infrastructure provides an abstraction layer on top of the available computational resources and environments, e.g. clusters and Clouds.

In order to enable self-scaling of processing units, XAP provides an advances monitoring service and Service Level Agreement (SLA) definition support based on the scaling rules specification. The monitoring service collects information about workload on the underlying computational resources, application availability and communication topology between processing units, as well as usage of business logic and data by applications. The user can utilize this information by defining thresholds for each monitored parameter value and specify actions which should be performed upon exceeding these thresholds. Moreover, thresholds can represent both maximum and minimum values of the monitored parameters.

XAP appears to be an attractive solution for building self-scaling applications. However, it is rather generic in scope and does not support data analysis methods or means for building data farming experiments. Moreover, it does not integrate with Grid environments as the underlying computational infrastructure. Regarding the

self-scaling aspect, the user can define strict thresholds but there is no support for trend detection or averaging historical measurements of the monitored parameters.

2.2.3 Teradata Database

One of the most important aspects of a data farming platform is efficient data management. In particular, storing large structured data sets can be challenging. This problem is not limited to the data farming methodology. One other example is data warehousing, which focuses on analysing data from a set of distributed databases. Traditionally, relational database management systems (RDBMS) were designed to work in the Online Transaction Processing (OLTP) mode, which often operates on single-row requests. Unfortunately, these databases, often perform poorly when faced with such operations as full-table scans, multiple-table joins, sorting or aggregating – all common data warehousing functions.

An important requirement of data warehousing solutions is scalability. This becomes especially important when building a data warehouse in an evolutionary manner, i.e. starting with a small installation and then extending it by adding new databases as data sources. Teradata Database [7] intends to provide a best-on-the-market solution for building data warehouses with near-linear scalability. The key aspect of Teradata Database is parallelization of query execution. By running multiple execution engines in parallel, each query can be processed much faster than in the traditional approach. However, to achieve this kind of processing, a shared-nothing approach must be adopted. Each physical node which belongs to a Teradata installation is responsible for handling a partition of data. Teradata takes care of distributing data to available physical nodes evenly.

The architecture of Teradata Database is depicted in Fig. 2.6. Each physical node hosts two types of components: Parsing Engine (PE) and Access Module Processor (AMP). Several instances of these components can run on a single physical node in parallel. Parsing Engines manage external connections to the system and perform query optimization. AMPs are responsible for managing a number of assigned rows and performing requested operations, e.g. manipulation, sorting, indexing and backing up. The last crucial element of this solution is the interconnect between physical nodes. Teradata provides a custom interconnect called BYNET for delivering messages, moving data and collecting results.

The scalability of Teradata Database is based on two aspects: shared-nothing architecture and scalable interconnect. Since components which reside on individual physical nodes are self-contained, they can be added to the system whenever necessary and operate in parallel. Each new node is assigned a portion of the overall dataset for management. However, this would be inefficient without a dedicated interconnect, i.e. BYNET, which utilizes several concepts (such as message aggregation, locality exploitation or column-based compression) to keep traffic to a

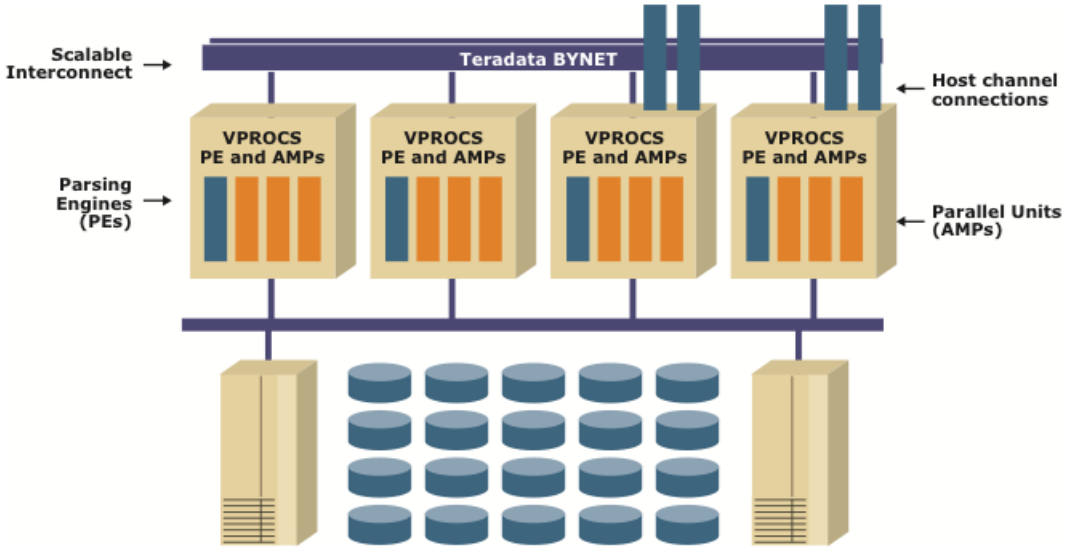

Figure 2.6: Deployment diagram of a TeraData installation [7].

minimum.

Unfortunately, Teradata Database is oriented towards data warehousing and therefore unsuitable for OLTP processing tasks. Moreover, it is based on many custom solutions, e.g. the BYNET interconnect, which increases the costs of such an installation. However, mechanisms which are utilized by Teradata Database to achieve massive scalability are generic and can be exploited in other systems.

2.2.4 Apache Hadoop

For a long time analysing large datasets with massively parallel tasks required in-depth expertise in the areas of hardware infrastructure, concurrency theory and building parallel applications. Many institutions, both academic or commercial, developed custom solutions to tackle this challenge. In most cases this resulted in immature, poorly scalable software which was forgotten as soon as the analysis was completed. However in 2004 Google proposed a simple programming model called "MapReduce" [74], which was internally used by the Google search engine. Under MapReduce the input dataset is divided into unrelated parts, each of which can be processed concurrently (Map phase). Following the processing phase all results are combined to form the output (Reduce phase).

MapReduce is merely a programming paradigm which has to be implemented by dedicated tools. One such solution, which has recently gained widespread popularity, is Apache Hadoop [75]. Although it began as a free implementation of MapReduce, it currently includes various subprojects for reliable, scalable, distributed comput-

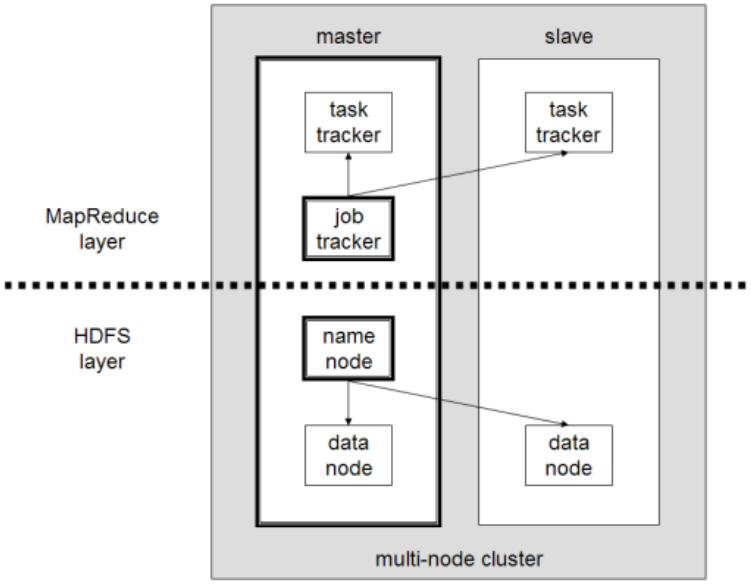

Figure 2.7: Simplified architecture of Apache Hadoop [8].

ing. Examples include HBase [76], which is a scalable, distributed non-relational database, the Hadoop Distributed File System (HDFS) [77], which provides high-throughput access to application data, and ZooKeeper [78], which is a coordination service for distributed applications.

The architecture of Hadoop is quite complicated and Fig. 2.7 only presents a somewhat simplified overview. Hadoop follows the master-slave pattern: the master is responsible for distributing tasks and collecting results from slaves which act as processing units. An important element of the architecture is the data distribution layer, which is implemented with HDFS. Similarly to Hadoop, HDFS follows the master-slave pattern, where the master node contains metadata about each stored file, along with its physical localization, and exposes a global namespace, while the slave node is responsible for storing the actual data. Hadoop maintains high operational efficiency thanks to data locality: during the map phase Hadoop dispatches tasks to nodes that are proximate to the data node which stores the necessary data.

The scalability of Hadoop is based on the elasticity of the master-slave pattern in terms of adding new slaves when necessary. As the HDFS layer follows the same pattern, it can also scale horizontally. Another important aspect is extendability. A common usage scenario is to first deploy Hadoop on a small cluster and then gradually extend it to hundreds of nodes as the application grows. The scalability of Hadoop and HDFS has been confirmed in many academic and commercial scenarios [79].

It should be noted that the size of a single Hadoop cluster is limited by the

capacity of its master node. In [79] the authors empirically estimate this to be approximately 4000 slave nodes. The only way to go beyond this number is to use multiple separate Hadoop clusters. As Hadoop is oriented towards batch processing, the cluster needs to be restarted to bring new resources online. Moreover, the master node can be treated as a single point of failure for the cluster. When it fails the whole cluster goes down. Thankfully, a number of possible solutions (e.g. the Facebook Avatar node citehadoop-avatar) have been developed to mitigate this problem.

2.3 Computational Environments

When building a massively scalable platform one should take into account executing applications in a heterogeneous computational infrastructure. In the following subsections we will describe the environments commonly used for large-scale application runs.

2.3.1 Grid computing

While conveying obvious advantages, distributed computing has also resulted in increased infrastructural complexity. Utilizing distributed computational resources to run applications can be a challenging task, especially for domain scientists who are not always IT experts. To facilitate this step several approaches and tools have recently been proposed, Grid computing being among the most important. The main idea of Grid computing is to render computational power and other related resources (e.g. storage or specialized devices) accessible in the same way as electricity. The term "Grid" was introduced in 1998 [80] to describe computational environments which possess the following properties (among others):

- coordinate resources that are not subject to centralized control,
- use standard, open, general-purpose protocols and interfaces,
- deliver nontrivial QoS.

Although the goal was well defined, it was not clear how such an environment could operate in practice. Several problems had to be solved – e.g. secure access to resources across institutional boundaries or ensuring the required QoS. Thus, various projects were initiated to study the issue further and develop the necessary tools.

In order to support modularity, most Grid projects proposed tier-based architectures similar to the one depicted in Fig. 2.8. The commonly encountered layers include:

- a resource abstraction layer, which is responsible for providing uniform access to various resources,
- a service layer, which includes all Grid-specific services supporting users applications,
- a security layer, which ensures that Grid resources such as computational power and sensitive data are not accessed by untrusted parties,
- a scheduling layer, which is responsible for deploying user jobs on the available resources.

Such a coherent set of Grid layers is often referred to as Grid middleware. It is often implemented as a software stack which turns distributed resources from many institutions into a coherent Grid environment. Note, however, that this deals merely with the technical aspect of the problem. The second part of constructing a Grid environment consists of procedures. As the Grid provides access to vast amounts of computational power, only trusted parties should be able to use it. Thus, it is necessary to uniquely identify each user of the Grid, in most cases via a personal certificate. The second characteristic feature is related to the way in which applications are running in the Grid. To create a single point of access to the environment Grids expose queuing systems to which users submit application (called Grid jobs in Grid parlance). The third interesting feature is the scientific orientation of Grid environments. Grids originated in the world of academia and remain more popular in the scientific community than in the industry. As a result, most applications running on Grid resources are scientific ones.

Although the concept of the Grid is simple, its implementation remains complex and poses many technical challenges. For several years the most common Grid user interfaces were (and arguably still are) command-line tools. GUIs are a minority, and in most cases focus on particular scientific disciplines. The second problem with Grids involves security measures which many users consider unintuitive and cumbersome. Thirdly, the diversity of Grid middleware makes the Grid quite difficult to use. As many Grid projects developed their own tools and services no standard middleware has emerged and there are still several divergent middleware packages in active use. Naturally, this precludes portability of Grid applications and services. Lastly, the Grid infrastructure is often less reliable than it should be. In most cases Grid middleware is powered by open-source software developed by volunteers from multiple countries. As a result, it sometimes contains insufficiently tested code.

Grid middleware

At the center of each Grid environment lies Grid middleware – an abstraction layer between Grid users and Grid resources. As presented in Fig. 2.8, Grid middleware

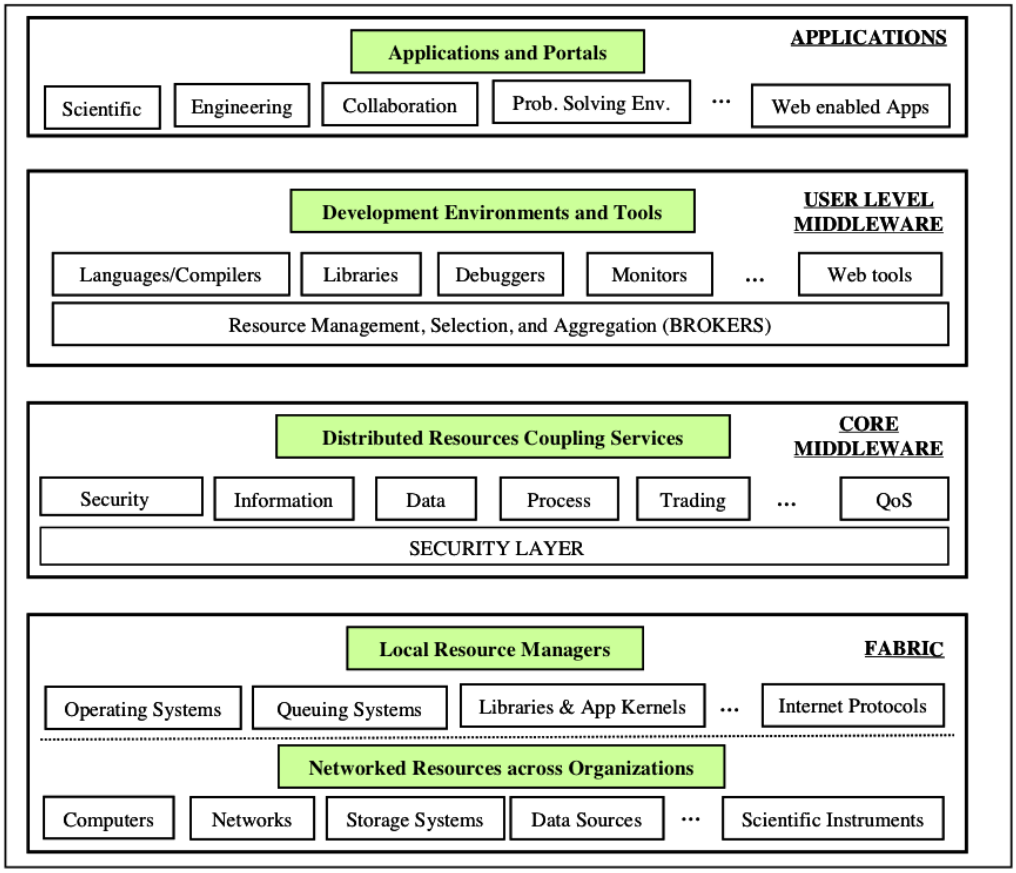

Figure 2.8: Tier-based overview of the Grid architecture [9].

consists of two sublayers: core middleware and user-level middleware. While the core middleware provides a uniform interface for accessing resources in the Grid Fabric layer, the user-level middleware is responsible for exposing services for Grid application developers, which provide high-level access to various Grid services, e.g. resource brokers or information services. A number of Grid middleware packages are available, each following a different approach to the common Grid objective.

UNICORE UNiform Interface to COmputing Resources (UNICORE) [81] is an integrated solution which facilitates seamless, secure and intuitive access to Grid resources, authentication mechanisms which can be integrated into administration procedures, and relocation of Grid jobs between different resources. UNICORE follows a tier-based architecture which divides Grid infrastructure into three types of elements. The first type, called the UNICORE Client, is responsible for preparation and monitoring of Grid jobs on behalf of the end user. It communicates, through a

component called the UNICORE Gateway, with the UNICORE Site (Usite) component, which represents a portion of the data center. Each data center can have one or more Usites. Each Usite offers access to Grid resources which are grouped into Virtual Sites (Vsites). UNICORE provides a Graphical User Interface to create and submit jobs. At the job creation stage the user can specify which files should be imported to worker nodes. These nodes are responsible for actually executing the job and exporting its results to a dedicated repository following completion. The user can describe jobs as a set of one or more directed acyclic graphs (DAGs). Upon submission each job is monitored and its current status reported to the user. In terms of security, UNICORE supports single sign-on through X.509 certificates.

Globus toolkit The second popular Grid middleware package is called Globus [64]. It provides a versatile open-source software toolkit which can be used to build Grid environments and Grid-oriented applications. Core Globus services enable remote access to distributed resources in a secure manner across institutional boundaries without sacrificing local autonomy. Globus implements three modules:

- Grid Resource Management, which enables resource allocation through job submission, staging of executable files, monitoring of job execution and collecting results. It integrates with various local schedulers such as Portable Batch System (PBS) or Load Sharing Facility (LSF). Jobs are defined using a dedicated language called the Globus Resource Specification Language. Any necessary files can be also specified in the job description and prefetched before the job is executed.

- Grid Information Services include a component called Monitoring and Discovery Service (MDS) which exposes an interface for registering and querying resource information. On each Grid resource a dedicated component (Grid Resource Information Service) is run, which responds to queries concerning resource properties.

- Grid Data Management, which defines an extension of the standard FTP protocol, namely GridFTP, and a replica management component. GridFTP enables efficient and reliable data transfer in a secure manner across Grid resources. The replica management components supports storing a single file at multiple sites and accessing it via a logical name, independent of the file's actual location. Detailed information about file replicas is available upon request.

QosCosGrid The third Grid middleware package worth mentioning is a relatively new Polish solution called QosCosGrid [82]. Its design was motivated by problems

with running complex simulations that can span hundreds or thousands of worker nodes, using common Grid middleware. QosCosGrid intends to provide resource reservation and fault tolerance capabilities. The architecture of the proposed solution divides the Grid infrastructure into so-called Administrative Domains (AD) which represent different data centers built on top of the Grid Fabric layer. QoSCosGrid leverages the queuing systems already available within each AD by providing a uniform remote interface (called the SMOA Computing service), supporting advance reservation of resources. Parallel execution of Grid jobs is supported by extending two popular environments: the OpenMPI [83] implementation of the Message Passing Interface standard for C/C++ and Fortran-based applications, and the ProActive framework [84] for running Java-based parallel applications. These extensions allow applications to communicate and synchronize between geographically distributed data centers. Additionally, the Data Movement component is provided to enable dataset prefetching. In order to submit jobs to ADs, a Grid Domain is necessary. The domain serves as an access point to the QosCosGrid infrastructure. Each Grid Domain exposes a Graphical User Interface for job preparation and submission, in the form of a Web portal. When a Grid job is ready for submission, the Grid Resource Management System (GRMS) is called to perform appropriate resource discovery and job management and monitoring (following actual submission). GRMS supports job definitions in the form of a workflow, with specific conditions applied to each workflow task. Moreover, GRMS has built-in support for parameter study jobs: the user can define parameter value ranges while GRMS automatically prepares and executes the necessary number of jobs, each with slightly different input parameter values.

2.3.2 Cloud computing

When planning and developing large-scale computational infrastructures, industry leaders such as Amazon, Google or Microsoft have always taken into account the highest possible number of clients whose requests may have to be processed. Thus, in most cases their data centers are underloaded, which can incur significant costs. This is an undesirable situation for a profit-seeking company and therefore many companies have opted to lease out the idle portions of their infrastructures to run third-party applications. In order to meet this goal they required a solution which would:

- be fairly easy to use by end users,
- reduce maintenance effort,
- come with a clear pricing model.

Existing Grid computing solutions suffered from poor penetration in the commercial market due to being overly complicated and unreliable. Thus, in 2005 Amazon Inc. introduced a new type of computational infrastructure, called Cloud [85], based on the idle portion of the infrastructure which powers the Amazon e-commerce portal.

The Cloud is often described as an unlimited pool of computing resources and storage space which can be accessed by users at the click of a button, with a pay-per-use model. This model enables users to avoid the high baseline costs of purchasing and installing actual hardware infrastructures. Instead, the Cloud provider maintains all hardware while the user obtains access to abstract, virtual resources, e.g. virtual machines with specified parameters. Moreover, by scaling upwards, large Cloud providers can offer very competitive pricing compared to in-house infrastructures. The underlying infrastructure is highly reliable and resource failures are usually transparent to end users.

In addition to raw computing power, Cloud providers offer a number of services which facilitate exploitation of the infrastructure. Common services include messaging, mailing, structural storage and scaling operations. While the former three are strictly application-oriented, the last one allows users to define how their applications should be scaled, both vertically and horizontally.

Another aspect of computational Clouds, of particular importance to this dissertation, is infrastructure scalability. When applications are run as Cloud virtual machines, the infrastructure can be scaled simply by adding new VMs. Even more importantly, the Cloud can provision new virtual machines in minutes instead of hours, which is especially important in highly dynamic environments. Moreover, some commercial Cloud providers expose services which enable starting and stopping Cloud virtual machines based on defined rules – for instance Amazon provides the Auto Scaling service for specifying scaling conditions and the CloudWatch service for monitoring virtual machine instances [86]. Microsoft Azure also allows the user to define rules based on which virtual machines are to be started and stopped [87]. However, both solutions are proprietary and focus on managing the infrastructure rather than actual applications (i.e. built-in scaling actions concern starting and stopping virtual machines running in the Cloud). In addition, both platforms are restricted to a single Cloud environment and do not support other types of infrastructures.

Although the Cloud may seem like a silver bullet for anyone wishing to lease computational resources, the approach is not without certain drawbacks. The most commonly raised issue involves security. As both data and applications reside on the Cloud provider's infrastructure, the user does not have full control over their management. They can be migrated between data centers and across national boundaries. This can be undesirable if the data in question is sensitive. The second aspect is

virtualization overhead, which may impact performance. Due to the highly heterogeneous nature of computational resources Cloud providers often introduce a virtual unit of computation which is used to describe different resources in a uniform manner. As a result virtual machines deployed in the Cloud will provide slightly inferior performance compared to physical nodes which are commonly encountered in Grid environments. Last but not least, the topology of connections between a given group of virtual machines can be far from optimal, rendering parallel computation less effective than in standard clusters.

Cloud taxonomies - deployment models and service models

Two of the most important taxonomies of existing Cloud computing systems focus on deployment models and service models respectively. The former is based on the visibility of a Cloud from the user's point of view. Public Clouds, which can be used by everyone without any constraints, possess the highest availability, although each Cloud infrastructure is owned by an organization selling Cloud services. This category includes Amazon Elastic Compute Cloud (EC2), Microsoft Azure [88], Google AppEngine [89] and many others. Private Cloud reside on the opposite end of the availability spectrum. In most cases they are limited to the resources of a single organization and can be accessed only from that organization's network, by authorized users. An organization can build a private Cloud using existing open-source Cloud stacks such as Eucalyptus [90], OpenStack [91] or OpenNebula [92]. Alternatively, a third party can provide an organization with a dedicated infrastructure for running a private Cloud. The third group, called hybrid Clouds, describes private Clouds whose computation power and storage capacity can be extended with public Cloud resources. A very important aspect of hybrid Clouds is technology that enables data and application portability between different Clouds. Unfortunately, due to the sluggish standardization process, portability and interoperability of various Clouds remain limited. The last group in this taxonomy is called community Clouds. They can be treated as an evolution of private Clouds, spanning several organizations which share a common goal. Such a Cloud is a single entity from the end user's point of view, but its infrastructure can be co-managed by several organizations. Community Clouds are often temporary, i.e. created to facilitate cooperation of several organizations for a specific purpose (such as a shared project).

The second important taxonomy concerns the manner in which the customer uses the Cloud, as depicted in Fig. 2.9. This taxonomy includes:

- Infrastructure as a Service (IaaS) Clouds, which provide access to a virtualized pool of resources (CPU, storage and networking) enabling customers to assemble virtual machines. The customer obtains access to the selected operating system and can deploy and run required software without any constraints.

While such low-level control is convenient and desirable, especially when deploying custom software, it requires additional effort on the customer's part. In this model the Cloud owner is responsible for providing reliable underlying infrastructure, i.e. physical devices and the interconnect layer.

- Platform as a Service (PaaS) Clouds provide access to well-defined runtime environments and programming services which can be used to develop applications without worrying about virtual machines. This model is much more convenient from the developer's point of view since it frees the developer from having to individually manage operating systems or physical infrastructure. The consumer is responsible only for providing an application developed using a software stack which is supported by the Cloud provider. On the other hand, the Cloud owner remains responsible for application deployment, availability and – in many cases – elastic scaling. In addition, the Cloud owner provides several services which facilitate the process of application development, such as (non-)relational databases, messaging middleware or e-mail facilities. In spite of existing constraints, especially concerning the available software stacks, PaaS Clouds offer an interesting developer-oriented alternative to IaaS Clouds.

- Software as a Service (SaaS) Clouds deliver specific applications which are deployed on the provider's infrastructure. SaaS applications are usually accessible through a thin client (such as a web browser) from arbitrary input devices. The Cloud owner manages the entire software stack, from the operating system to the provisioned application. In this model the consumer focuses on using provided application rather than on developing his/her own applications.

Open-source Cloud solutions

Historically, the first Cloud solutions were the proprietary, closed, commercial platforms operated by large enterprises such as Amazon, Microsoft and Google. Although they were public and could be used by anyone, only the Cloud owner knew how the Cloud worked internally. From the consumer's point of view such a Cloud was a black box. Since 2005 (the year Amazon EC2 was introduced) a number of new Cloud computing solutions have been launched. Some of them remain commercial in nature, but others are open-source products. Thanks to open-source Cloud middleware many companies and research facilities can run a private Cloud using private resources. While most of these deployments are oriented on performing computations and did not support any form of "Data as a Service", modern open-source Cloud solutions often provide rudimentary data storage features.

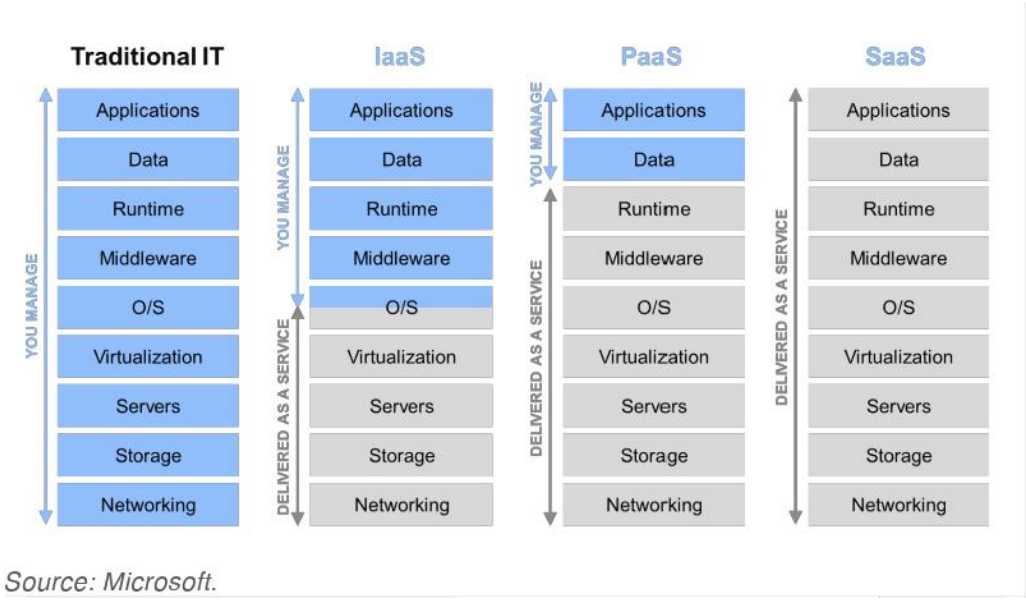

Figure 2.9: Taxonomy of Cloud service models [10].

Eucalyptus The first successful open-source Cloud system which provides storage on demand was Eucalyptus [90]. Introduced in 2008 at University of California, Santa Barbara, Eucalyptus is an example of an open-source project which became very popular outside the scientific community. Many commercial entities currently develop their own private Clouds using Eucalyptus. In fact, Eucalyptus is often treated as a model IaaS Cloud solution. Regarding functionality, Eucalyptus aims to provide an open-source alternative to the Amazon EC2 Cloud. It exposes a programming interface to its services (virtual machine management and storage), which is compatible with programming interfaces exposed by the Amazon EC2 Cloud. Each Eucalyptus installation consists of several loosely coupled components, each of which can run on a separate physical machine to increase scalability. The architecture of Eucalyptus is depicted in Fig. 2.10. The frontend of an Eucalyptus Cloud is the Cloud Controller element, which is an access point to virtual machine-related features. While the Cloud Controller is responsible for computations, the Walrus component handles data storage. It can store virtual machine images along with any other files, organized into a hierarchy of buckets and can be used in the same way as Amazon S3. Each virtual machine is executed on a physical host, which is controlled by the Node Controller element. A group of nodes can be aggregated into a cluster which exposes a single access point (the Cluster Controller for virtual machine management and the Storage Controller for access to the virtual machine image repository).

There are two versions of the Eucalyptus Cloud: Community and Enterprise.

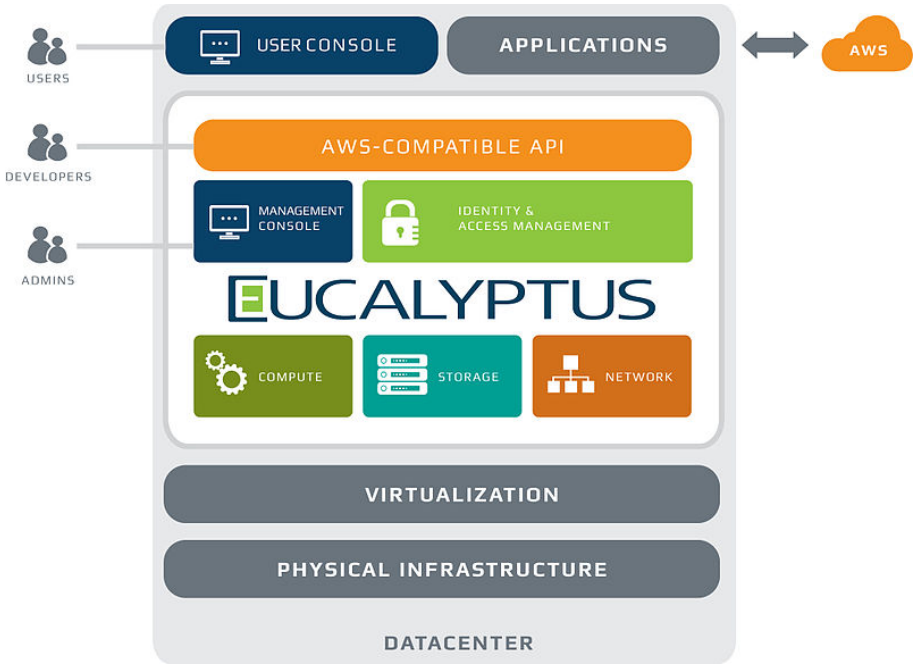

Figure 2.10: Architecture of the Eucalyptus Cloud [11].

The free version of the Eucalyptus system stores data in a single directory on the host on which the Walrus component is installed. As such, the only way to distribute data is to exploit a distributed file system, e.g. Lustre or Oracle Cluster File System 2, mounted in the directory used by the Eucalyptus installation. The file system remains orthogonal to the Cloud solution, i.e. it does not have access to any information about the Cloud and can only manage data based on some basic properties such as the size of stored files the or capacity of the available storage resources. Such strategies are very limited and are cannot be easily adapted to suit the Cloud computing paradigm. The problem is tackled by the Enterprise version of Eucalyptus which, among other features, provides an adapter for direct integration with Storage Area Networks (SANs), e.g. Dell Equallogic or NetApp. However, to the best of our knowledge, this integration does not allow combining different types of storage systems within a single Cloud installation. Moreover, the Cloud administrator cannot declare a policy for distributing data among the available storage resources. Data management therefore devolves upon SAN, which knows nothing about the Cloud, its users or the type of data stored in the Cloud. Although SANs are enterprise-class data storage solutions, they do not provide any Cloud-specific storage strategies which would acknowledge e.g. information about Cloud customers.

OpenStack In 2010 NASA and RackSpace jointly launched the OpenStack Cloud initiative, which intends to enable any organization to create an IaaS Cloud on commodity hardware. NASA contributed to the project by releasing its middleware, called Nebula [92], for managing virtual machines at physical infrastructure. In turn, RackSpace contributed its storage solution known as Cloud Files [93]. Since the launch, over 120 companies have joined the OpenStack project, including Intel, AMD, HP, Dell Cisco and Citrix Systems. OpenStack is a collection of tools for building a virtual infrastructure using resources available to a data center. In terms of computations, OpenStack provides the OpenStack Compute (Nova) solution, which is responsible for provisioning and managing instances of virtual machines. To control the OpenStack-based Cloud both graphical and programming interfaces are provided, compliant with Amazon EC2 and Rackspace Server APIs.

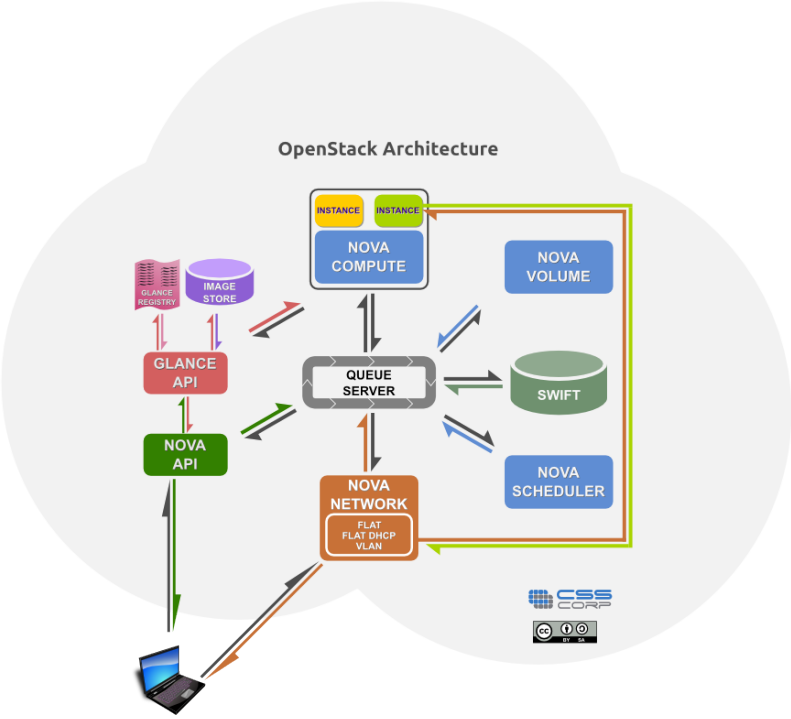

Figure 2.11: Architecture of the OpenStack solution [12].

In terms of storage, OpenStack provides OpenStack Object Storage (Swift), which is an object storage solution with built-in redundancy and failover mechanisms. Each stored object is transparently replicated to multiple hardware devices. Swift handles failover situations automatically by ensuring the replication level of each stored object. In addition, it supports dynamic scaling of the underlying storage by adding or removing storage resources. The programming interface of OpenStack Swift is compatible with Amazon S3, thus any existing application which uses

Amazon S3 can also leverage OpenStack Swift. There is also a separate subsystem, called OpenStack Imaging Service, responsible for managing virtual machine images. OpenStack was designed with scalability in mind. Thus, its internal architecture, depicted in Fig. 2.11, uses the shared-nothing, messaging-based approach and each of its major components, i.e., Cloud Controller, Volume Controller, Object Store and Network Controller, can be run on multiple servers. Communication between components is based on asynchronous method calls via HTTP and AMQP to avoid blocking.

Although OpenStack provides many important features regarding high-availability data storage, it lacks mechanisms for increasing data access performance or differentiating data on the basis of its purpose. Each stored object is treated in the same way, regardless of its importance or intended use (note that data which is frequently updated should be replicated less eagerly than data which is only read). Moreover, OpenStack does not provide any monitoring service which would observe and analyze user behavior with respect to data usage.

Task farming

Executing a large number of independent tasks, often referred to as task farming or bags of tasks, is a popular approach while using HTC systems, e.g. Grids and Clouds. Furthermore, this approach is often applied as a component of simulation-based scientific workflows [94]. By enabling parameter studies, task farming can be considered part of the data farming process, as described in Chapter 1.

Although task farming can be conducted in any computational environments, using Cloud environments is especially interesting due to the availability of various types of resources at limited cost. Utilization of Cloud environments to conduct task farming is a popular research topic [94]. Contrail is an EU-funded project which intends to design, implement, evaluate and promote an open-source system for Cloud Federations [95]. As a part of the Contrail project, the ConPaaS software stack is developed [96], which addresses the problem of porting existing applications to the Cloud. The project aims at supporting familiar programming models so that existing applications can be easily migrated to the Cloud. To achieve this goal ConPaaS provides services which act as replacements for commonly-used runtime environments, e.g. MySQL databases or PHP runtimes.

Another product of ConPaaS is related to task farming and its main component is a budget-constrained scheduler called BaTS [13]. The main objective of the BaTS scheduler is to minimize the cost of running a bag of tasks using Cloud resources by allocating the most efficient resources for the given task. Cloud environments such as Amazon EC2 offer different resources at different costs. BaTS estimates the required budget for the given bag of tasks by evaluating the efficiency of executing tasks using different resource types.

The BaTS architecture is depicted in Fig. 2.12. Task execution is divided into two phases. The first phase, called the sampling phase (the left side of the figure), is related to budget estimation. BaTS schedules a small number of tasks on different types of resources and calculates the total cost for each resource type. Based on the estimated budgets, ConPaaS allocates Cloud resources. The second phase (the right side of the figure) is called execution and involves execution of the remainder of tasks from the initial bag. During this phase online monitoring is used to refine the initial execution plan, if necessary.

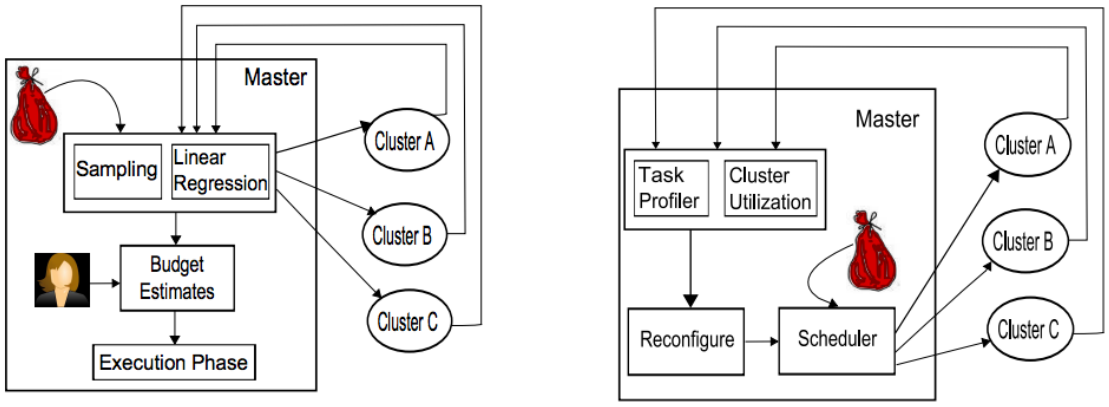

Figure 2.12: A budget-constrained scheduler architecture [13].

Chapter 3

Massively Self-Scalable Platform: Concept and Architecture

In this chapter the author describes the development process of a virtual platform for data farming starting with use cases along with their functional and non-functional requirements. Next, the author considers adapting existing software architectures to match the design of the proposed platform. Due to the limitations of existing architectures the author introduces an extension to SOA in the form of self-scalable services which addresses the scalability non-functional requirement. Based on the proposed architecture, an overview of the platform's design and a description of the required functional modules are presented.

3.1 Development Methodology for a Data Farming Platform

Although this thesis addresses scientific and engineering problems affecting massively self-scalable software, it was inspired by an actual problem related to the development of a virtual platform for data farming. We decided to utilize the waterfall model as the main software methodology since all functional requirements were well known beforehand. Only two development iterations (i.e. building a prototype version and building the final version) were planned, precluding the need for more agile methodologies such as Scrum.

The waterfall process starts with a requirements analysis phase which is often supported by use case definition and specification of requirements. Each use case refers to a complete scenario performed by the client using the platform to achieve a meaningful effect, e.g. creating a new data farming experiment. Each use case may involve a number of high-level actions performed be the client, e.g. specifying input parametrization followed by application of DoE methods. Finally, each high-level action can be translated into a number of low-level operations performed both on the client and server sides, e.g. specifying a single input parameter or calculating the

expected number of simulations based on currently selected parametrization criteria.

Once all use cases have been defined, a design phase is carried out by extracting functional blocks and discussing the available architectural styles suitable for the platform. Analysis of existing architectural styles such as layer-based, service-oriented and space-based architectures, is presented later on in this chapter. Due to limitations regarding scalability, the author proposes an architecture style based on the *self-scalable service* concept, which is among this dissertation's key contributions. This new architecture style intends to address the scalability requirement as a first-class citizen by combining concepts known from existing architectural styles and concurrency programming models. The author then applies these concepts to describe the platform architecture.

The next phase of the waterfall process is implementation. In the context of the presented platform implementation details are discussed in Chapter 5. Platform evaluation, which is the next phase of the process, is described in Chapter 6. The final phase of the process, i.e. maintenance, is described in Chapter 7.

3.2 Platform Use Cases

Use cases allow us to define the main goals of software in terms of functionality which should be provided from the end user's point of view. In addition, end users need to be specified as part of this phase. The author intend to build a virtual platform for data farming, which addresses problems encountered by analysts and decisionmakers (often seen as the main beneficiaries of the data farming methodology). Thus, the virtual platform aims at supporting different phases of the data farming process. An overview of the supported use cases is presented in Fig. 3.1.

The supported use cases can be divided into two groups, based on the actor who executes each use case:

1. data farming use cases, depicted on the left side of Fig. 3.1, including activities related to preparation and execution of data farming experiments,

2. platform management use cases, depicted on the right side of Fig. 3.1 and related to platform operation and maintenance.

The following subsections summarize the supported use cases.

3.2.1 Data Farming Use Cases

This group includes use cases related strictly to each phase of the data farming process which was described in Chapter 1. In particular, these use cases cover such features as preparation of a data farming experiment, monitoring its progress and

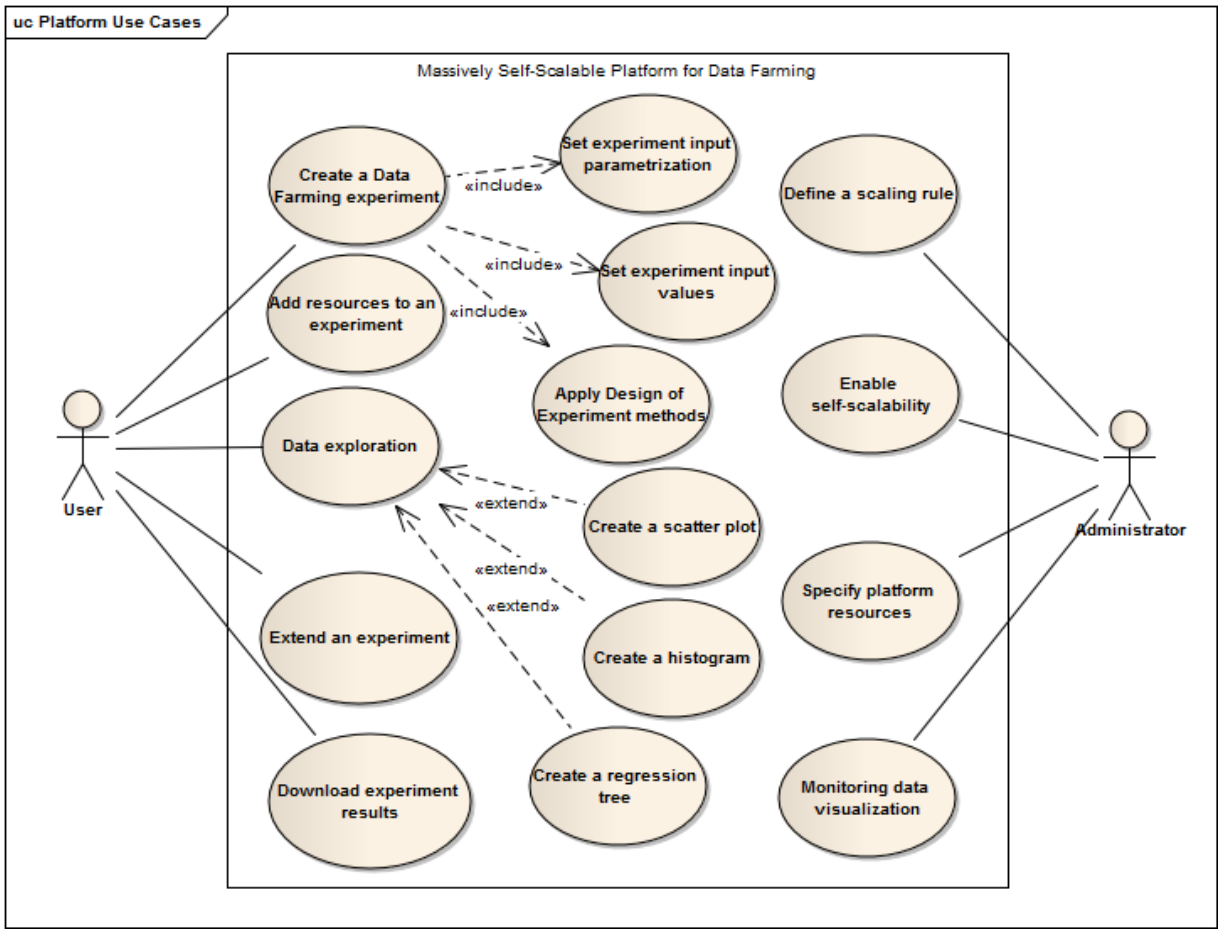

Figure 3.1: A use case diagram for a virtual data farming platform.

exploration of data with built-in mechanisms. A summary of each use case in this group is presented below (cf. left side of Fig. 3.1):

1. *Create a data farming experiment* involves the first three steps of the data farming process, as depicted in Fig. 1.1, namely 'Experiment objectives definition', 'Simulation scenario building' and 'Input space specification'. During this use case the decisionmaker specifies all attributes of a data farming experiment:

   - a simulation model, which will be used during simulation execution,
   - a parameter space to explore during the experiment, including parametrization methods, input value constraints and DoE methods,
   - an initial set of resources which will be used by the simulation.

2. *Add resources to an experiment* is related to the 'Simulation execution' phase of the data farming process. Although the "master" part of the platform, depicted in Fig. 1.3, is self-scalable, the user can adjust the amount of computational resources dedicated to the "worker" part dynamically. Having access to different infrastructures, e.g. Grids and Clouds, the user can decide how many resources should be used by the platform to run workers and execute simulations included in the data farming experiment, e.g. in terms of CPU and RAM. In most cases the source of computational power should be transparent to the user, i.e. starting new virtual machines in a private infrastructure or submitting jobs to the Grid environment should be equally transparent. However, if public Clouds (e.g. Amazon EC2) are required, the user should be able to select this type of infrastructure explicitly and acknowledge the associated financial conditions.

3. *Data exploration* is related to the 'Output data exploration' phase of the data farming process. In general, simulation output data can be explored in various ways, e.g. using visualizations or data mining methods. It is difficult to implement all possible options in a single platform and therefore we have decided to include only a few visualization methods (described in more detail in Chapter 7), which facilitate basic analysis of simulation results:

   - scatter plots for pairs of MoE and/or input parameters,
   - histograms of MoE values,
   - regression tree graph.

4. *Extend an experiment* relates to a transition between 'Output data exploration' and 'Input space specification'. Nowadays the practice of conducting data farming experiments involves creating several small-scale experiments to investigate a given phenomenon, and then following up with large-scale studies to thoroughly explore interesting subspaces. However, such an approach is error-prone due to the need for separate data farming experiments focusing on different parts of the input parameter space. The presented platform intends to eliminate this process by enabling users to create an experiment which can later be extended to cover additional input parameter subspaces in an evolutionary way. By extending a single experiment in different directions the user can compare simulation results from different subspaces more easily than with separate experiments.

5. Downloading experiment results (experiment management group) is a necessary step for further analysis of results with third-party tools as well as sharing them with people who do not use the platform. The result of an experiment can

be divided into two parts. The first part contains information about each simulation run, i.e. input arguments and resulting MoE values. This data should be accessible in a commonly used standard, e.g. a CSV file, supported by by third-party statistical analysis tools such as JMP [97] or R [98]. The second part of the experiment's result includes more detailed information about each simulation run, i.e. actions performed within the simulation. Although this information is simulation-specific, it can be useful e.g. when tracing potential errors.

3.2.2 Platform Management Use Cases

An important feature of the presented platform is massive self-scalability, which calls for a set of features related to platform management. A summary of the use cases related to this aspect is presented below (cf. the right side of Fig. 3.1):

1. *Define a scaling rule* is a basic operation related to platform self-scalability. The main goal of scaling rules is to express expert knowledge about platform scaling behavior. The platform administrator can define rules for each platform service to enable self-scalability.

2. *Enable self-scalability* is a simple switch which can be set by the administrator. The platform will use monitoring data along with scaling rules to scale itself once the self-scalability feature is enabled.

3. *Specify platform resources* is an operation related to selecting resources which comprise the master part of the platform. Selected resources are used during scaling operations to start new manager instances.

4. *Monitoring data visualization* is an administrative tool which provides administrators with information about the current and historical platform workload. This information can be utilized to define new scaling rules or modify existing ones.

3.3 The Massive Self-Scalability Requirement

The previously described use cases do not cover non-functional aspects of the platform which address issues related to QoS, platform maintenance and user experience. In the context of a data farming platform several such requirements can be discerned:

1. massive scalability to support large-scale data farming experiments,

2. simulation execution with heterogeneous infrastructures, i.e. local clusters, Grids and Clouds (whether public or private),

3. cost-effective resource utilization through self-scalability.

Massive scalability is necessary to execute thousands of simulations in parallel. As explained in Chapter 1, data farming experiments often require computational power which exceeds the capacity of a single data center. This translates into the need for heterogeneous infrastructures. Self-scalability is an essential feature which contributes to optimal resource utilization.

Self-scalability can be defined as *the ability of a platform to automatically adjust the quantity of resources used in response to emerging events*. Scale adjustment can have a different meaning depending on the platform, e.g. allocating an additional processor in a multiprocessor system or using additional servers to run applications. In either case, scale adjustment involves changing the set of resources assigned to the platform. The "self" prefix denotes that the adjustment is performed automatically, without human interaction. However, conditions, which trigger this action should be defined beforehand, typically by domain experts. The platform itself is responsible for detecting whether such conditions have actually occurred, and for initiating the scale adjustment procedure.

To cope with the need for massive self-scalability we have identified the following additional functional requirements:

- ability to represent expert knowledge regarding scaling conditions and procedures,

- collecting monitoring information describing the workload of each platform element,

- execution of multiple instances of each platform element, maintaining location transparency,

Concerning the first requirement, both aspects, i.e. specification of conditions and self-scaling procedures, need to be represented in a formal way to be processed by computer systems. In order to address this need, the author introduces the concept of scaling rules in Chapter 4.

The remaining two requirements are indirectly related to platform architecture and can be met by way of appropriate design. The platform should be designed as a set of elements which fulfill the following criteria:

1. each platform element is a unit of modularization which provides the desired functionality through a well-defined interface,

2. platform elements communicate with each other directly and do not form a hierarchy,

3. each platform element can be deployed multiple times with each instance using a different set of physical resources; however all instances of a given platform element should expose a single access point (if necessary),

4. for each platform element, scaling conditions and procedures can be defined separately.

Many different architectural styles are applied in modern IT design. The most popular ones include:

1. *Layer-based architecture* [99], where functionality is provided by tiers, each of which has a well-defined interface and responsibilities. Each tier communicates only with tiers located directly below and above itself. The user interacts with the topmost tier, while the lowest tier is usually associated with physical hardware. Arguably the most common incarnation of this architecture comprises three tiers: data access, business logic and presentation.

2. *Service Oriented Architecture* (SOA) [100], which assumes that each application can be assembled from a set of loosely coupled services with well-defined interfaces. Hence, each service is a separate peer and can be replaced easily. An important feature of SOA is interoperability, i.e. the ability of services to work together to provide the desired functionality. SOA promotes flexibility and adaptability of applications by encouraging developers to create reusable and well-focused services, which can be easily utilized in multiple applications.

3. *Space-based architecture* [101] is an architectural style for achieving linear scalability by dividing applications into processing units, i.e. self-sufficient modules, each of which provides the full functionality of an application. Each processing unit is independent and the application can be scaled up/down simply by adding/removing processing units. Note, however, that the requirement of distributing an application as a monolithic module may result in significant overhead as applications typically contain components responsible for data access, business logic, messaging, presentation etc.

The first two architectural styles intend to cope with software complexity by introducing modularization and separation of concerns. They do not address the problem of scalability directly as this problem was not deemed critical when these architectural styles were being introduced. However, along with the growing popularity of the Internet in the late 90s and in the early 21th century, applications had to face the problem of handling rapidly varying number of clients (e.g. due to

a sudden surge in the popularity of a given application). Hence, these styles do not fulfill all of the necessary requirements.

The third architectural style, i.e. space-based architecture, was introduced to solve the scalability problem by using self-sufficient processing units. A processing unit can be instantiated multiple times on different computational resources to provide the necessary performance. However, it is assumed that each processing unit encapsulates all the features of an application, which can impose significant overhead when considering unbalanced components (for example a single database component can handle multiple business logic services). Overall platform performance depends on the load of each component which may vary from request to request – a single request can involve many data access operations or many CPU-intensive operations. In such cases components may require different scaling conditions, which is not supported by space-based architectures.

To fulfill all the presented requirements, an extension to SOA called Self-scalable services is proposed in the next section. This extension aims to provide all the necessary features for designing massively self-scalable platforms.

3.4 The Concept of Self-Scalable Services

To the best of our knowledge, none of the existing architectural styles fulfill all the requirements described in the previous section. In particular, these architectural styles do not address the massive self-scalability of data farming platforms. On the other hand, each of the presented styles provides important and desirable features:

- separation of concerns in the layer-based architecture,
- loose coupling in SOA,
- self-sufficient modularization units in the space-base architecture.

As a consequence, *this dissertation proposes an extension to SOA called self-scalable services, which is a set of service instances grouped together, along with additional shared modules providing self-scalability* . This extension represents a combination of the above mentioned features in the form of a self-scalable modularization unit. A data farming platform has been developed to verify the applicability of the proposed extension to building highly scalable software.

When considering massively self-scalable platforms composed of loosely coupled services, the most challenging requirement is management of multiple instances of each service running in parallel in such a way that all instances are treated as a single unit by external clients. To address this requirement we cannot operate on the level of services (treated as individual instances). Instead, we have to extend the

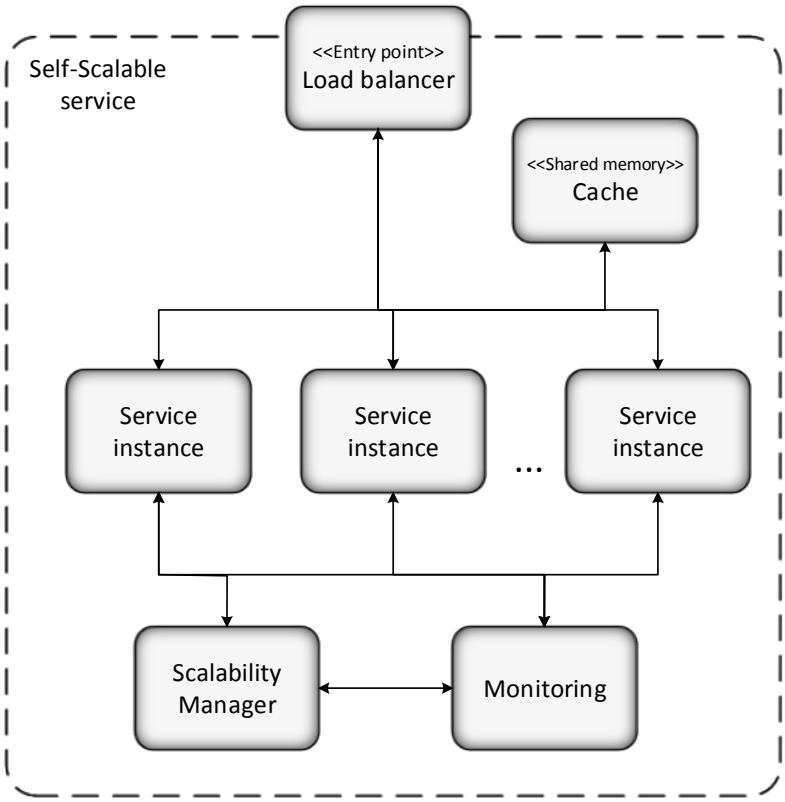

Figure 3.2: Overview of a self-scalable service.

meaning of a service as a software modularization unit to embrace the scalability feature. We therefore introduce the concept of a *self-scalable service*, which can be defined as a group of service instances sharing the same functionality, accessible as a single unit by external clients.

An overview of a self-scalable service structure is depicted in Fig. 3.2. It is an independent modularization unit which enriches any service with the self-scalability feature. Each such service contains the following elements:

- *one or more instances of a service* denotes an original service which provides the desired functionality,
- *load balancer* forwards requests from external clients to service instances and constitutes an entry point for the self-scalable service (though each service instance can be accessible directly if necessary),
- *cache* service, which can be utilized as shared memory for all service instances enabling communication between them and providing a common place to store seldom-changing data utilized by service instances,

- *monitoring* system, which collects information about the self-scalable service workload,
- *Scalability Manager*, which is responsible for enforcing scaling rules in an automatic way.

Crucial self-scalability features are provided by the last two elements, i.e. the monitoring service and the Scalability Manager. To achieve the desired QoS (e.g. mean response time less than some arbitrary value) these elements utilize the *feedback loop* approach, popular in control systems and autonomous computing. Monitoring is a sensor which provides information about service workload, while the Scalability Manager is a controller which adjusts the number of concurrent service instances according to the observed workload. Scale adjustment is based on defined scaling rules which express expert knowledge about the desired scaling behavior. Scaling rules are provided to the Scalability Manager of a self-scalable service by the administrator who manages the self-scalable service.

3.5 Self-Scalable Services in the Data Farming Platform

To demonstrate the capabilities of self-scalable services we have built a platform called Scalarm, which stands for Massively Self-Scalable Platform for Data Farming. Scalarm is a complete multi-tenancy platform for data farming, which implements all phases of the data farming process, starting from experiment definition through simulation execution to result analysis. In addition, Scalarm enables users to manage computational resources assigned to simulations regardless of their source – this includes private resources, Grids and Cloud environments.

Considering the explicit scalability requirement, self-scalable services are especially well suited for designing the platform. The architecture of Scalarm is depicted in Fig. 3.3. Each service except the Information Manager is self-scalable by design, i.e. several common modules are utilized alongside actual service instances.

In the context of the platform four services are defined:

1. *Experiment Manager* handles all interaction between the platform and end users via a graphical user interface. It also constitutes a gateway for analysts, providing a coherent view of all running and completed data farming experiments and enabling analysts to create new experiments or conduct statistical analysis of existing ones. Finally, the Experiment Manager is responsible for scheduling simulations using Simulation Managers.

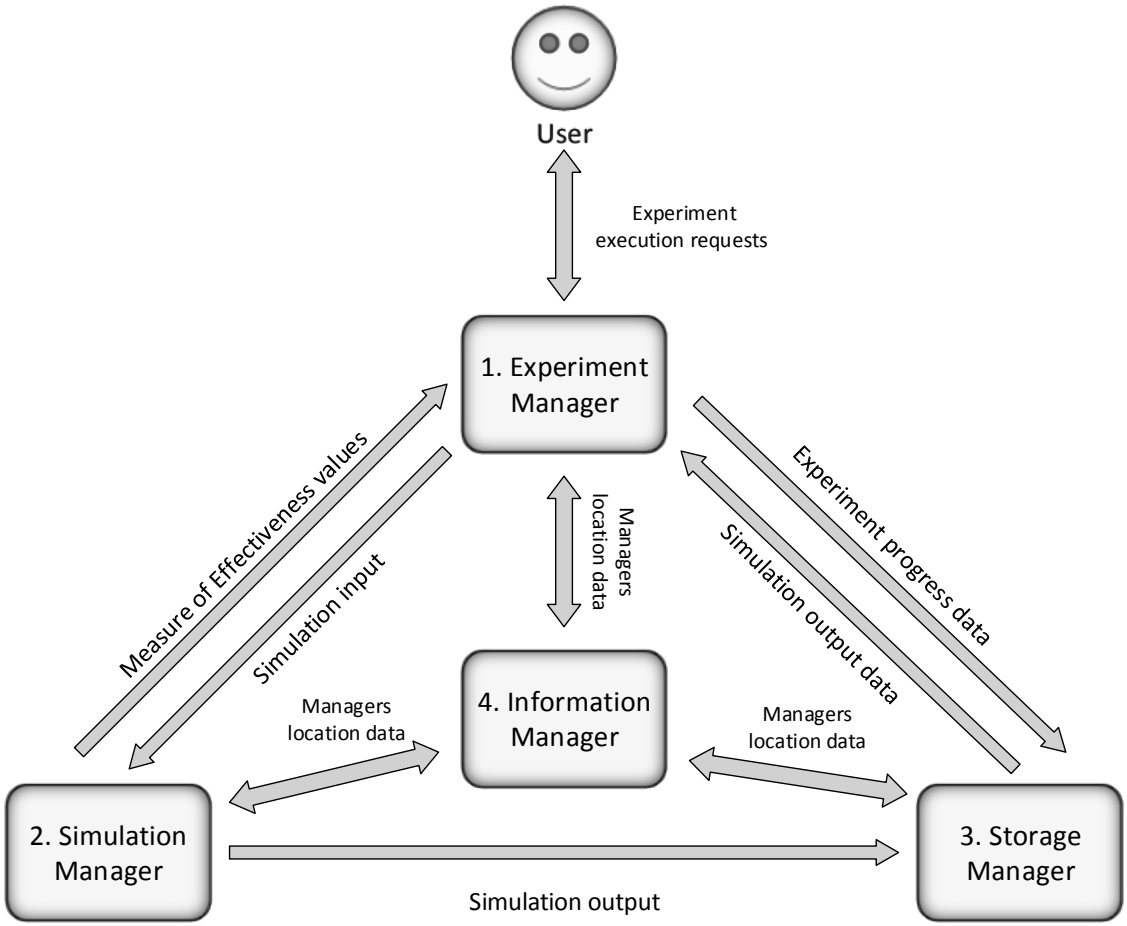

Figure 3.3: High-level overview of the Scalarm architecture.

2. *Simulation Manager* is a wrapper for actual simulations which can be deployed in various computational infrastructures, e.g., private clusters, Grids or Clouds. It can be treated as an implementation of the pilot job concept, i.e., a special application that acquires computational resources to run applications. While the pilot job concept was devised specifically for Grid environments, the Simulation Manager is infrastructure-independent. The wrapper is responsible for preparing the simulation environment, e.g. by downloading the necessary code dependencies and input parameter values. When the simulation concludes the Simulation Manager uploads its results to the "master" part: log files and other binary datasets are sent to the Storage Manager while MoE values are sent to the Experiment Manager along with a notification that the simulation has completed. Since it operates in a highly dynamic and unreliable environment, the Simulation Manager is guarded against Experiment and

Storage Manager failures as well as network connectivity issues. Moreover, to maximize resource utilization, the Simulation Manager may schedule multiple simulations in parallel based on computational resource capabilities, i.e. additional simulations are started as long as they do not significantly impact the performance of simulations which are already in progress.

3. *Storage Manager* implements the persistence layer concept in the form of a separate service. Other components, particularly Experiment and Simulation Managers, use this service to store different types of data: structural information about each executed simulation and experiment, and results of simulations (which may be either binary or textual). By utilizing a built-in load balancer, the Storage Manager can be treated as a virtually centralized but physically distributed single data storage point, facilitating client-side operations while preserving performance and scalability.

4. *Information Manager* is an implementation of the service locator pattern, known from SOA-based systems. It is a "well-known" place for each component in the system, which stores information about other components' locations.

The first three services need to be massively self-scalable to enable Scalarm to operate on a large scale. In light of this requirement they were designed as self-scalable services. The fourth service – the Information Manager – is less frequently used and therefore has no need for self-scalability, though it was designed as a highly available service using similar concepts (multiple instances governed by a load balancer).

Chapter 4

The Problem of Scalability

In this chapter the author describes the scalability feature in more detail. First, the need for scalability is outlined, followed by a description of the most popular scalability metrics. Common scaling strategies and potential bottlenecks in web applications are described and the concept of scaling rules is introduced to represent expert knowledge about scalability management, which is the second main contribution of this thesis. Finally, the scalability of the Scalarm platform is discussed.

4.1 Motivation for Scalability

As highlighted in Section 1, scalability has only recently become the defining feature of modern large-scale systems. At first, scalability referred to the extendability of computing infrastructures [102]. This was especially desired when designing a large installation which would need to expand in the future. The rise of parallel algorithms gave new meaning to the scalability concept – it was now interpreted as the ability of an algorithm to utilize additional resources, e.g. CPU or memory, when available [103]. Designing and implementing such algorithms was, and still remains, the main goal of scientists who have access to large-scale computer installations. In order not to reimplement the same scientific algorithms for each new machine, these algorithms had to become scalable. This meaning of scalability has recently been subsumed by the domain of distributed applications, especially web-oriented ones. From a business point of view, it is necessary to have applications which can cope with increased workload (generated by additional clients) simply by adding more computational power. This approach, i.e. increasing the available computational resources on demand, is one of the main benefits of Cloud computing. In this context, a new aspect of web application scalability is the ability to scale upward as well as downward, taking into account the cost of running web applications in large businesses. Using all the available resources to run an application all the time may incur high costs as most resources remain idle. By the same token, combining scalable applications with flexible computational infrastructures can be very cost effective since the amount of allocated resources varies along with the actual workload.

Throughout this dissertation the *scale* of a computer system, in particular the Data Farming platform, refers to the amount of resources it uses. In the context of self-scalable services the scale of a service can be measured by the number of instances running in parallel, which indirectly influences the amount of utilized resources. In addition to scalability, we can define *scalability management* as a set of management operations related to application management in the context of scale adjustment, i.e. discovering when the application should be rescaled and performing the actual rescaling. Appropriate scalability management is required to achieve efficient scalability in self-scalable systems. Unfortunately, there is currently no standard way of expressing expert knowledge regarding scalability management.

4.2 Scalability Metrics

To measure the scalability feature a number of different metrics have been proposed. All of them reflect changes in system properties corresponding to changes in scale. The system scale represents the amount of resources – e.g. processors, storage or memory – available to the system. Scalability of computational algorithms can be measured with regard to the number of processors. On the other hand, scalability of a software platform requires a different scale, representing e.g. the number of active servers. For the purpose of this thesis we will use the general concept of a "compute unit" in order to represent system scale.

The most widely used metric for describing scalability, called *Speedup*, "S", compares the total execution time for a program running on a single compute unit to execution time obtained with N compute units. It can be calculated using Eq. 4.1.

$$S(N) = \frac{T_S}{T_N} \tag{4.1}$$

where T_S is the execution time of the fastest known serial algorithm and T_N is the execution time with N compute units. This particular form of speedup is called *fixed size speedup* as the execution time in either case is measured with respect to the same total workload. In a perfectly scalable system speedup equals N, i.e. regardless of the amount of compute units work can be divided into N parts and performed in parallel.

In 1967 Gene Amdahl suggested that the maximum speedup of some types of applications depends on the portion of the application that cannot be parallelized [104]. An important prerequisite is that the application must execute the same number of instructions for the same input data regardless of whether processing occurs in a serial or parallel fashion. Amdahl's law is mathematically represented by Eq. 4.2.

$$S(N) = \frac{T_S}{T_N} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}} \tag{4.2}$$

where s is the non-parallelizable fraction of the application while p is the parallelized fraction of the application. For the sake of convenience we assume that $s + p = 1$.

Amdahl's law assumes that the problem size remains identical in both runs, which is hardly ever true. Instead, one may fix the time slot in which the application is run and instead increase the size of the problem. This scenario is described by Gustafson-Barsis's law and expressed by Eq. 4.3.

$$S(N) = \frac{s + p * N}{s + p} = N + s * (1 - N) \tag{4.3}$$

where s is the non-parallelizable fraction of the application. For the sake of convenience we assume that $s + p = 1$.

As both laws produce different results, many viewed Amdalh's law as incorrect. However, in [105] Yuan Shi claimed that both laws are, in fact, consistent. In the Amdahl's formula the non-parallelizable fraction of the algorithm is independent of the number of compute units, as expressed by Eq. 4.4:

$$s_{ns} = \frac{t_s}{t_s + t_p(1)} \tag{4.4}$$

where s_{ns} is the the non-parallelizable fraction of the algorithm in the Amdahl's formula (often referred to as non-scaled percentage of the serial part program), t_s is the processing time for the non-parallelizable fraction of the algorithm using a single compute unit and $t_p(1)$ is the processing time for the parallelizable fraction of the algorithm using a single compute unit.

In Gustafson's formula this fraction is dependent on the number of compute units, as expressed by Eq. 4.5:

$$s_s = \frac{t_s}{t_s + t_p(N)} \tag{4.5}$$

where $t_p(N)$ is the processing time for the parallelizable fraction of the algorithm using N compute units. s_s is often referred to as scaled percentage of the serial part program. Both values are related by Eq. 4.6.

$$s_{ns} = \frac{1}{1 + \frac{(1-s_s)*N}{s_s}} \tag{4.6}$$

where s_{ns} is the non-scaled percentage of the serial part program while s_s is the scaled percentage of the serial part program.

The second metric, i.e. efficiency (Eq. 4.7), E, describes how additional resources are utilized by a parallel version of an application. The value of efficiency is commonly thought to fall in the $[0, 1]$ range, however practice shows that it is possible to achieve efficiency greater than 1, e.g. by storing all of the application's data in RAM instead of on a hard drive. Efficiency is often used to categorize the type of scalability provided by an application, with three distinct groups: linear (efficiency equal to 1), sub-linear (efficiency lower than 1) and super-linear (efficiency greater than 1).

$$E(N) = \frac{S(N)}{N} \tag{4.7}$$

Scalability can be expressed either using the above mentioned metrics or with dedicated formulas. There are two popular scalability metrics. The first one is based on efficiency, and therefore called *Efficiency-based scalability*, "S_E", as expressed by Eq. 4.8.

$$S_E(N_1, N_2) = \frac{E(N_2)}{E(N_1)} \tag{4.8}$$

The second scalability metric invokes the concept of *productivity* introduced in [106] to combine system performance with the cost of computation. Taking cost into account is especially important when using commercial Clouds for computing. *Productivity* is the capability of a system to process incoming requests which should be maintained at different scales to call the system scalable. The authors define the productivity of a "request-response" system using the following properties:

- $\lambda(k)$ - system throughput in responses per second at scale k,
- $f(k)$ - abstract value of each system response, called *response value*, indicating QoS at scale k; it be expressed e.g. by the mean response time of the system,
- $C(k)$ - cost of the system at scale k, indicating the total quantity of different types of resources such as compute units, memory and disk space allocated by the system.

The productivity metrics (F) can be then expressed according to Eq. 4.9:

$$F(k) = \frac{\lambda(k) * f(k)}{C(k)} \tag{4.9}$$

Using these properties, *Productivity-based scalability*, "S_F", can be calculated with Eq. 4.10:

$$S_F(k_1, k_2) = \frac{F(k_2)}{F(k_1)} = \frac{\lambda(k_2) * f(k_2) * C(k_1)}{\lambda(k_1) * f(k_1) * C(k_2)} \tag{4.10}$$

In real-life applications scalability can depend on multiple factors and should be studied with different metrics reflecting different perspectives. Scalability analysis of an application requires in-depth understanding of the system behavior, going beyond the raw values of scalability-related metrics. Such analysis will be performed in Chapter 6 with regard to the presented Data Farming platform.

4.3 Common Scaling Strategies and Potential Bottlenecks

In the context of web applications two scaling strategies can be distinguished: scaling up (vertically) and scaling out (horizontally). Scaling up means adding more resources to a machine which runs the application. In this strategy we manipulate the resource pool of a single machine, e.g. adding more CPUs or more RAM. A scalable application should discover and utilize any available additional resources automatically. It is worth noting that scaling up can be easily achieved by most web applications simply by exploiting the local OS and its inbuilt mechanisms such as threads and virtual memory.

On the other hand, scaling out (horizontally) means adding new resources to the application in the form of additional machines. In this scenario the application obtains all crucial resources, i.e. CPU, memory, disk and network, as a coherent unit. Compared to scaling up, scaling out can provide much more total computational power to an application. However, the application itself has to be designed differently to operate in a distributed environment. It has to be able to discover new available machines and utilize them. This approach promotes a decentralized approach to software design, i.e. there should be no single resource required for the application to operate.

To enable large deployments which adjust themselves to changing workload patterns with minimal administrative intervention, the platform has to provide near-linear scalability along with high performance. The latter is necessary in any production platform in order to minimize the resources required to handle a single client. On the other hand, scalability is needed to cope with increasing workload.

Based on the above-mentioned rules governing scalability, i.e. Amdahl's and Gustafson's laws, we can conclude that a massively scalable platform can be developed only by minimizing the non-parallelizable fraction of algorithms executed by the platform. This includes any action performed by the platform starting from experiment generation, through simulation scheduling and execution all the way to result analysis.

In traditional layer-based web applications, scalability bottlenecks typically involve the database tier [107]. There is often a single database which persists state for

the entire web application, while the business logic layer provides stateless services to ensure scalability. Modern relational database management systems are mature, efficient and can handle multiple clients simultaneously but they still constitute a single point of failure. To increase reliability and availability, it is possible to run two or more instances of a replicated database management system. To increase scalability, data representing application state has to be sharded and distributed to more than one server.

Another bottleneck of software platforms is related to the load balancing of connections between clients and platform services, where each service and each service instance is deployed on a different physical server. When considering several platform services and their instances, along with a much higher number of clients, it is crucial to balance the workload evenly between the available servers. There are several common load balancing strategies:

- DNS load balancing [108] involves using the DNS system to bind several IP addresses to a single domain name and then return addresses in a round robin manner to successive clients. It does not require any modification either on the service or on the client side. However, it is less flexible than other methods since it does not take into account information about service workload.

- Client-side load balancing is a strategy in which the client is aware of all available service instances. For each request the client can select a different service instance to interact with. This strategy is the most scalable, because each client is responsible for individually selecting a service instance. However, this can lead to suboptimal resource usage since clients are not aware of the current load conditions on the service side.

- Using a dedicated component called a load balancer, which handles all incoming requests and redirects them to different service instances based on a set of rules. The load balancer can access monitoring information about each service instance and can provide optimal workload balance in terms of resource utilization. However, due to being a single access point, it cannot scale linearly in large installations.

The final bottleneck which should be considered in the scope of the Data Farming platform regards data storage of simulation output. Depending on the scenario, each simulation can generate several megabytes of data, which means that a single experiment can produce terabytes of data, or more. This amount of data is more than a single hard drive can store, hence a dedicated storage system, such as a disk array, should be used. As management component instances will be distributed among multiple servers, this storage system should be accessible via a network and

capable of handling large-scale write throughput from multiple clients. These requirements are hard to meet with a single storage system. Thus, a more scalable system is required.

4.4 Scaling Rule Definition

In this section, the author introduces the concept of *scaling rules* as another contribution of this dissertation. Scaling rules express the intended scaling behavior of a self-scalable service; in particular – how and when service instances should be started and stopped.

Self-scalable services intend to facilitate development of self-scalable platforms such as Scalarm. An important element of such a platform is automation of scaling procedures which can be achieved by representing knowledge about scalability management (i.e. when and how to scale the given platform) in a computer-processable form. In computer science, a well-known form of expressing knowledge involves rules [109]. Such rule-based knowledge is often consumed by domain-specific expert systems [110], which support decision making by end users.

While expressing knowledge in the form of rules is commonplace in modern systems, there is no standard way of specifying rules for scalability management. Such dedicated rules should describe conditions which will trigger scaling actions, in addition to scaling strategies and metrics. Scaling rules intend to fill this gap by providing a machine-processable way for defining conditions, metrics and actions concerning scalability management. A scaling rule can be defined as the following tuple:

$$ScalingRule := < Metric, MeasurementType, Condition, Threshold, Action >$$
$$MeasurementType := SimpleMeasurement \mid TimeWindowMeasurement$$
$$Condition := < \mid > \mid ==$$
$$Threshold := SimpleValue \mid PercentageValue$$

where the tuple elements are as follows:

- *Metric* denotes any measurable parameter of a service included in the scalable system, e.g. CPU utilization level or service response time,
- *MeasurementType* indicates the way in which a service parameter is measured. *SimpleMeasurement* gets the most recent measured value of the parameter, while *TimeWindowMeasurement* aggregates measurements over a given period of time.
- *Condition* is a logical operator which defines the desired relationship between the Metric value and the Threshold.

- *Threshold* is a numerical or percentage-based value of the Metric, which, combined with the Condition operator, determines when a scaling action should be triggered.
- *Action* denotes what should be done when a rule is met (e.g. add new computational resources or shut down a service instance).

Multiple scaling rules can be combined into a single compound rule using logical operators:

$$CompoundScalingRule := ScalingRule \mid ScalingRule\ OR\ CompoundScalingRule \mid ScalingRule\ AND\ CompoundScalingRule$$

By using these scaling rules, a self-scalable service can be precisely described in terms of conditions governing up- and downscaling. For each such a service a set of compound scaling rules can be defined, which will encapsulate scaling management knowledge. As a result, scaling rules should be adapted to the characteristic features of the service, e.g. high consumption of RAM. Scaling rules can be grouped into three categories, based on the combination of measurement types and thresholds:

- Simple scaling rules, which include rules with $SimpleMeasurements$ as the measurement type and $SimpleValues$ as the threshold type, e.g. “service response time is less than 100 ms.”
- Time window rules, which include rules with $TimeWindowMeasurements$ as the measurement type and $SimpleValues$ as the threshold type, e.g. “average response time over the last 5 minutes is less than 100 ms.”
- Trend discovery rules, which include rules with $TimeWindowMeasurements$ as the measurement type and $PercentageValues$ as the threshold type, e.g. “response time increased by more than 200 % of the mean value over the last 5 minutes.”

4.5 Scalability in the Scalarm Platform

In the context of the Data Farming platform, vertical and horizontal scalability of the "master" part (depicted in Fig. 1.3), is desired for the following reasons:

- Executing data farming experiments of varying sizes - each experiment can include a different number of simulations, from dozens to thousands or more. Hence, the platform needs to enable adjusting the quantity of computational resources at runtime.

- Platform multi-tenancy - the number of clients interacting with the platform at any given moment can vary significantly. To maximize the resource utilization level, the platform should reserve the minimal amount of resources required to handle the current workload and be able to allocate additional resources when the workload increases.
- On-demand boosting of experiment execution - the user may want to execute simulations involved in the experiment as fast as possible, depending on the experiment's priority. Hence, the platform should enable the user to manually adjust the amount of resources allocated to simulations.
- Resource partitioning between "masters" and "workers" - executing experiments can result in variable workload to which the platform should adjust even when dealing with a fixed amount of resources. The platform should partition the available resources between "masters" and "workers" according to the ratio of work handled by each type of component respectively.

We utilized self-scalable services to address the above mentioned issues during the design phase of the platform. Each self-scalable service is a fully independent modularization unit which can be managed independently to obtain the required scalability. The Scalarm platform is divided into four services, as described in Section 3.5. Each service has different resource requirements:

- *Experiment Manager* handles actions related to the three main phases of a data farming experiment:
  - input space specification,
  - output data exploration,
  - progress monitoring and simulation scheduling as parts of the simulation execution phase.

  The first two phases are mostly CPU- and memory-bound as computations need to be performed on possibly large datasets. The third phase is mostly network-bound because it requires querying databases in the *Storage Manager*.
- *Storage Manager* is responsible for persisting information about data farming experiments performed by the platform. This information comes from two different sources:
  - information about each simulation included in the experiment, e.g. input parameter values, current status and results. This data is well suited for structural storage systems, such as (non-)relational databases.

  - binary results of completed simulations, e.g. log files, which can be used for further analysis.

  The volume of binary results can be much larger than the volume of ancillary information. This calls for a different type of storage system: one which provides limited performance but is more cost-effective. Activities performed by these components are primarily I/O- and memory-bound.

- *Simulation Manager* handles actual simulation execution based on input parameter values obtained from *Experiment Manager*. Once the simulation is finished, MoE values and any binary results are sent to *Storage Manager*. Depending on the simulation, these components can be CPU-, memory- or I/O-bound.

- *Information Manager* is responsible for storing configuration information required by other self-scalable services (e.g. location of access points for each self-scalable service). This information represents shared knowledge about the platform, required for initialization of new service instances and integrating them with existing services. The associated resource requirements are negligible.

Utilizing self-scalable services provides the option to attach different scaling rules to different services. This feature follows the SOA principle of building systems from loosely coupled and independent services. Actual scaling rules for each self-scalable service can vary depending on the specific deployment, e.g. physical server capabilities or communication layer properties. The following key factors are taken into account when considering different self-scalable services:

- Scaling rules for *Experiment Manager* should involve parameters describing the overhead of simulation execution, e.g. simulation scheduling time. In addition, metrics related to data exploration can be defined, such as the preparation time for a specific type of graph.

- Scaling rules for *Storage Manager* should be related to parameters describing I/O performance, e.g. read and write operations per seconds and the average completion time for I/O requests, including the time spent in queues.

- Scaling rules for *Simulation Manager* should involve parameters describing the current utilization level of basic machine resources, e.g. CPU and RAM.

Although detailed scaling rules will be defined on the basis of experimental evaluation presented in Chapter 6, we can specify some preliminary scaling rules based

on the above mentioned key scaling factors. We begin by defining a set of scaling actions, which includes only two operations, namely starting a new component instance and stopping an existing instance:

$$Actions = (\ StartNewInstance, StopInstance\ )$$

We also define a set of monitoring metrics which includes information about the monitored system properties:

$$Metrics = (\ AverageResponseTime,\ AverageNumberOfRequestsPerSec,\ AverageNumberOfReadOperations,\ FreeDiskMemory,\ CpuUtilization\ )$$

By using the presented actions and metrics we can define sample scaling rules for each main self-scalable service. These rules are listed in Table 4.1. Each self-scalable service can utilize different metrics to decide when and how to scale. In addition, different measurement types can be used in different metrics.

Table 4.1: An outline of sample scaling rules for defined self-scalable services.

Service	Rule objective	Metric	Measurement type	Condition	Threshold	Action
Experiment Manager	scale up	Average Response Time	TimeWindow = 1 minute	>	100ms	Start New Instance
Experiment Manager	scale down	Average Number Of Requests Per Sec	TimeWindow = 1 minute	<	1	Stop Instance
Storage Manager	scale up	Free Disk Space	TimeWindow = 1 minute	>	10%	Start New Instance
Storage Manager	scale down	Average Number Of Read Operations	TimeWindow = 1 minute	<	30	Stop Instance
Simulation Manager	scale up	CPU Utilization	TimeWindow = 30 seconds	>	98%	Start New Instance
Simulation Manager	scale down	CPU Utilization	TimeWindow = 30 seconds	<	40%	Stop Instance

Chapter 5

Scalarm Implementation Details

In this chapter the author describes essential aspects of the implementation of the Scalarm platform. First, the services constituting the platform are described in detail. This is followed by a description of key elements of the platform architecture, implemented to support massive scalability. The self-scalability feature is discussed in terms of its implementation. Finally, behavior of the platform in typical situations is described, including creation of a new data farming experiment, executing simulations and extending an experiment.

5.1 Platform Overview

Scalarm is a complete multi-tenant platform for data farming, which facilitates all phases of the data farming process, starting with experiment definition, through progress monitoring to analysis of results. In addition, Scalarm enables users to manage computational resources regardless of their source, e.g. private resources, Grids or Cloud environments. The basic design pattern which underpins the Scalarm architecture is "master-worker", hence internal components can be divided into two groups: managers and workers. The former group handles actions related to logical organization of computation and infrastructure management, while the latter handles actual execution of simulations.

The main use cases and (non-)functional requirements of Scalarm are described in Chapter 3. Due to the massive self-scalability requirement, Scalarm utilizes the concept of self-scalable services to organize its architecture, i.e. Scalarm functionality is provided by a number of independent services, each of which scales itself and exposes a single access point on demand (if necessary). While existing systems for data farming or simulation scheduling are only capable of scaling workers, Scalarm supports scaling of managers as well, to maximize the resource utilization level.

In terms of scalability management, Scalarm utilizes the concept of scaling rules introduced in Chapter 4. Scaling rules express expert knowledge on scaling each platform service. Combined with information about the current state of the platform (obtained from a dedicated monitoring system), they constitute the self-scaling

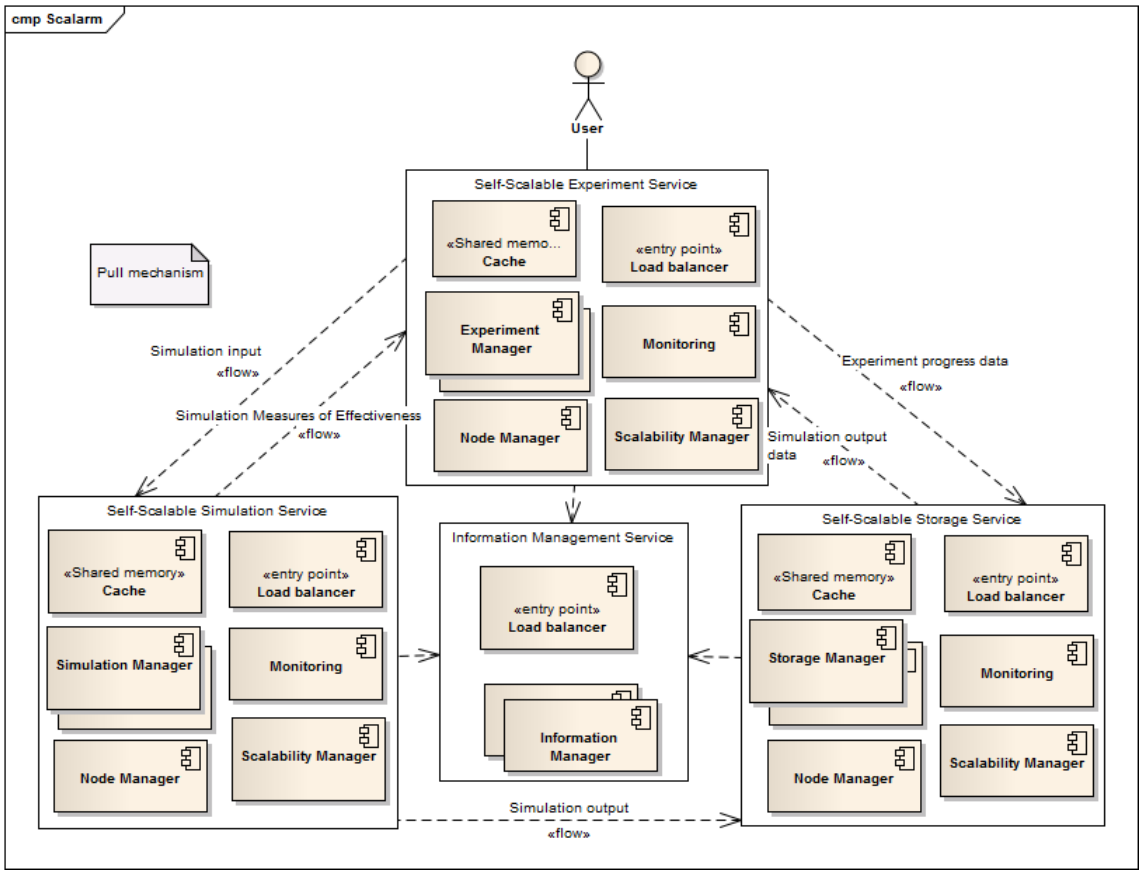

Figure 5.1: Component diagram of Scalarm.

feature of Scalarm. In Chapter 4 several scaling rules are outlined in the context of Scalarm services to demonstrate the flexibility of the concept. In addition, Scalarm enables adding new scaling rules on demand to adjust scalability management for a specific deployment to suit user requirements.

5.2 Scalarm Services

The architecture of the Scalarm platform, introduced in Chapter 3, is depicted in Fig. 5.1. Key self-scalable services were previously described in terms their features and resource requirements. In this section the author extends this description with implementation details for each service. In addition, other modules are introduced and described – these modules do not provide any end-user features but are essential for the platform itself.

Services utilized in the Scalarm platform can be divided into three groups:

- *Data farming*-related services, which support all phases of the data farming process. This group includes Experiment, Storage and Simulation Managers.
- *Platform maintenance*-related services, responsible for managing the platform as a whole. This group includes Information and Node Managers.
- *Common services* utilized by self-scalable services and included in each such service. This group includes Monitoring, Scalability Manager, Load balancer and Cache.

The following subsections discuss the implementation of each of the above mentioned services.

5.2.1 Experiment Manager

Each GUI-enabled virtual platform requires a dedicated service to handle all interaction with end users. In Scalarm this task falls to the Experiment Manager which exposes a uniform GUI. It also constitutes a gateway for analysts, i.e. provides a coherent view of information concerning all running and completed data farming experiments, enabling analysts to create new experiments and conduct statistical analyses of existing ones. The Experiment Manager is a manager-type component for Simulation Managers, i.e. it is responsible for scheduling simulations and retrieving results.

Its internal architecture is depicted in Fig. 5.2. The Graphical User Interface is provided by two components, i.e. 'Experiments' and 'Infrastructure'. The former provides an interface for creating, manipulating and extending data farming experiments, enabling statistical analysis of partial results. The latter provides an interface for managing data farming infrastructure, e.g. adding computational power to speed up execution of a specific experiment, if requested. To communicate with other components, e.g. Storage Managers, the Experiment Manager utilizes lookup information about components' entry points from the Scalarm Information Service via the 'Configuration' component. Information about currently running and historical experiments is partially stored in a shared database within the self-scalable service that supervises Experiment Managers. Information concerning actual simulations within each experiment is stored in a non-relational database handled by Storage Managers. To enable multi-tenancy of Scalarm, a dedicated component called 'Users' handles authentication, authorization and user management.

The Experiment Manager provides a Graphical User Interface (GUI) for analysts to conduct and monitor data farming experiments. This GUI is divided into two logical parts: experiment management and infrastructure management. Each part provides multiple views associated with different activities. Control flow within the experiment management (important from the user's point of view) is depicted

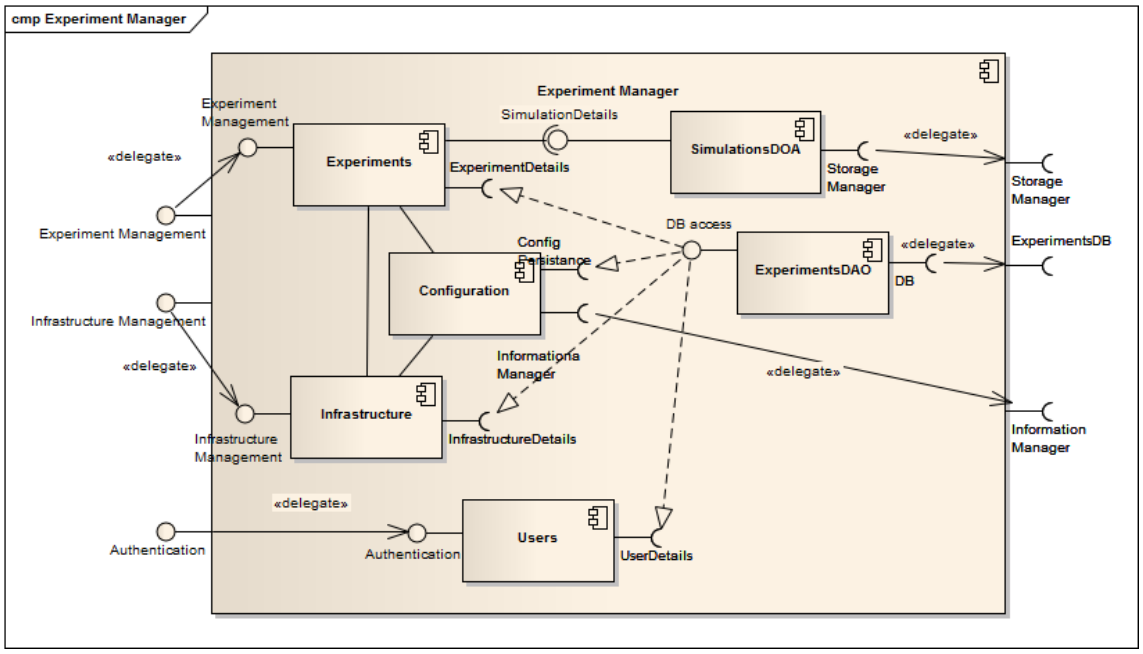

Figure 5.2: Internal architecture of the Experiment Manager.

in Fig. 5.3. The User Interface aims to focus all user experience on data farming experiments. Thus, upon login, the user is redirected to a monitoring view showing the most recently started experiment, or to a list of available experiments (if no experiment is currently running). From the monitoring view, the user can perform different actions related to experiment management. If an experiment has only recently been started and there is no data to analyze, the user can assign additional computational power from resource pools, e.g. Grids or Clouds, to execute the experiment. Following execution of some simulations the analyst may perform statistical analysis of partial results using built-in statistical analysis mechanisms, e.g. regression trees or histograms. At all times the user can display the monitoring view, showing any ongoing or completed experiments, e.g. to compare results. In addition, new experiments can be prepared and started using various DoE methods. More information on this aspect will be presented in Chapter 7 in the context of Scalarm support for training of security forces.

5.2.2 Storage Manager

In traditional tier-based applications, data storage is handled by a dedicated tier, called "persistence" or "database". It stores all the necessary information in a non-volatile manner and provides an access interface adapted to a specific application. It is, in turn, utilized by the business logic tier as a data access layer, and is of-

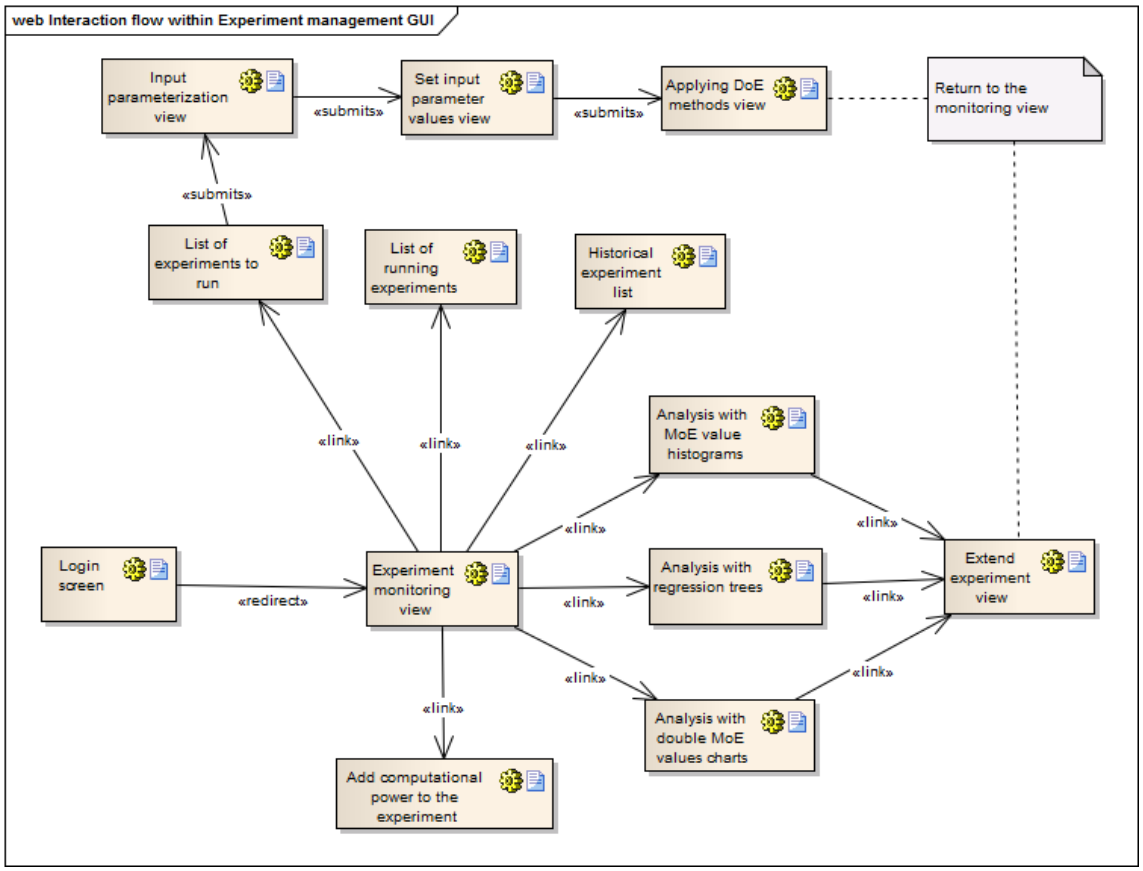

Figure 5.3: Interaction flow with Experiment Manager using the provided GUI.

ten implemented as a relational database. Other tiers, e.g. the frontend layer or clients, know nothing about it and cannot directly use it to store any additional information (in line with the principles of tier-based architectures). Such approach provides separation of responsibilities between tiers and conceals implementation details within an abstraction layer. On the other hand, it can prove constraining since all storage-related operations have to be handled by the business logic layer.

Scalarm utilizes the persistence layer concept in the form of a separate service called the Storage Manager. Other components, mainly Experiment and Simulation Managers, use this service to store different types of data: structural information about each simulation and experiment, and results which may be either binary or textual. By utilizing the concept of self-scalable services, the Storage Manager can be treated as a virtually centralized but physically distributed single point of data storage, supporting clients while preserving performance and scalability.

The internal structure of the Storage Manager is depicted in Fig. 5.4. Interfaces provided by this service enable storage of text/binary results from simulations

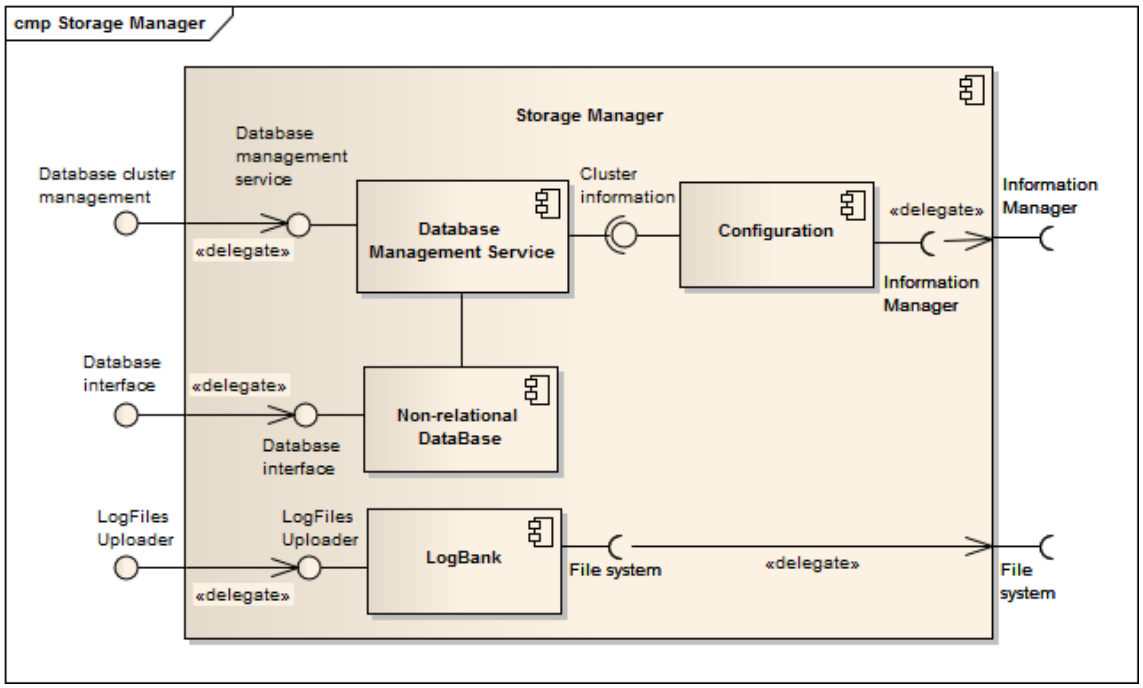

Figure 5.4: Internal architecture of the Storage Manager.

(e.g. log files), persisting information about experiment progress in a non-relational database, and handling scaling requests. The latter interface is needed as the platform relies on third-party solutions such as a non-relational database.

The current version of the manager utilizes the MongoDB non-relational database [111] as it provides clustering and scaling features. MongoDB also implements scaling-out mechanisms, which are consistent with the concept of self-scalable services, i.e. transparent clustering of multiple instances with a single access point to the cluster. Furthermore, MongoDB supports data sharding of a single table within a cluster, i.e. rows stored in a single table are transparently distributed among the available resources.

For binary/text log file storage, the "LogBank" component utilizes standard filesystem mechanisms. It can connect to either local or remote/distributed storage resources, such as a disk array. The prototype is meant to work with a disk array shared by multiple Storage Manager instances using the Network File System (NFS) protocol [112]. This greatly simplifies implementation of the component, but constraints its scalability, due to NFS limitations. However, this was not an issue in real use cases where Scalarm was utilized.

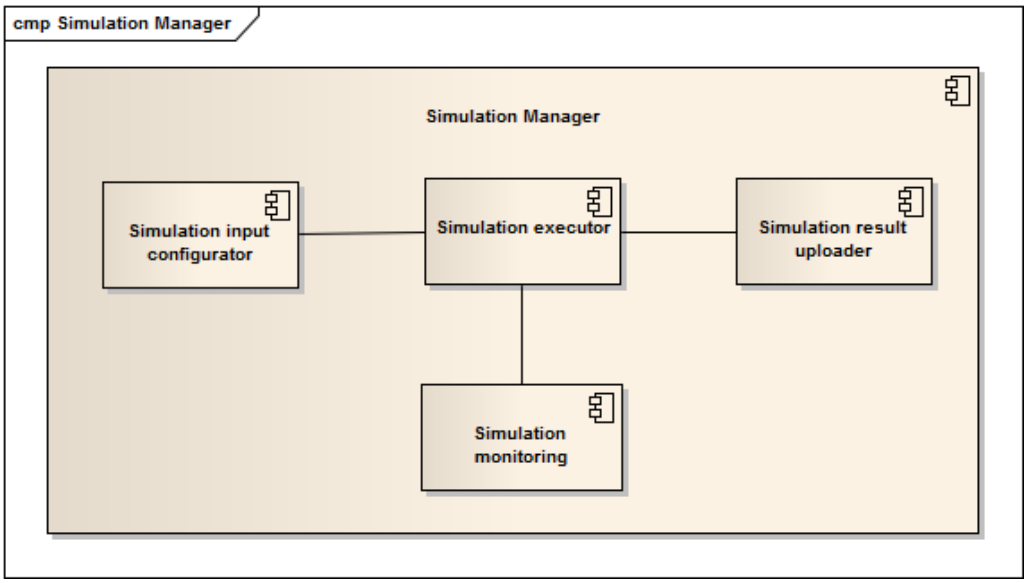

Figure 5.5: Internal architecture of the Simulation Manager.

5.2.3 Simulation Manager

At a high-level of abstraction, Scalarm follows the "master-worker" pattern, i.e. there are services (masters), which schedule simulations to and obtain final results from other services (workers). This pattern separates execution of simulations from management of data farming experiments. Furthermore, the platform infrastructure can be divided into resources dedicated to management of simulations and resources which actually execute simulations. In Scalarm the "worker" part of this pattern is implemented by the Simulation Manager. All Simulation Managers are logically grouped into another self-scalable service, though individual instances can run on physically distributed resources. They also do not need a single entry point because no other components communicate with them. Assuming usage of heterogeneous computational infrastructures, e.g. Grids and Clouds, it is hardly possible for each utilized computational resource to be publicly accessible on the network. Thus, to eliminate this requirement, the Simulation Manager is implemented as an active service which initializes communication with other services.

This service is a wrapper for actual simulations, which can be deployed on various computational infrastructures, e.g. private clusters, Grids or Clouds. Its internal structure is depicted in Fig. 5.5. The Simulation input configurator component is responsible for preparing the whole environment for a simulation, i.e. downloading the necessary code dependencies and input parameter values. Simulation execution is handled by the Simulation executor. Once the simulation concludes, the result uploader sends results (log files and MoE values) to the "master" part. To max-

imize resource utilization the Simulation executor can start multiple simulations in parallel, depending on the capabilities of the computational resources which are monitored with the Simulation monitoring component. With such a wrapper actual simulations can be prepared and tested locally by simulation domain experts and no Scalarm specific code needs to be added.

The Simulation Manager can also be treated as an implementation of the pilot job concept [67], i.e. a special application that acquires computational resources to run actual applications. However, while the pilot job concept was devised for Grid environments only, the Simulation Manager is infrastructure-independent, i.e. in theory it can work with any present or future computational infrastructure. To verify this statement, prototype Simulation Managers have been prepared for private infrastructures, Grids and Clouds. To maximize resource utilization, Simulation Managers prefetch configurations for subsequent simulations using the *pull* mechanism which increases scalability by removing the need to store information about each worker.

5.2.4 Information Manager

In highly distributed and dynamic systems a very important aspect is auto discovery. New instances of services can be added to the system in an unpredictable manner. Each newly added service has to be configured to be aware of other component types and their access point locations. To facilitate this, SOA-based systems utilize the Service locator pattern [113], where a designated service is responsible for registering all other exposed services. By applying this pattern services need to be aware of only one location from which current information about other services can be collected.

The Information Manager is an implementation of this pattern in Scalarm. It is "well-known" to each component in the system and stores information about the location of other services. Owing to self-scalability, Scalarm services can obtain endpoint access information for any specific service they require. During instance startup, e.g. following a scale-up action, the instance communicates with the Information Manager to obtain information about dependent services – for instance, Experiment Managers inquire about Storage Manager access points while Simulation Managers request locations of Experiment and Storage Managers. In addition, the Information Manager preserves the sources of other services, which are used by the Scalability Manager to enact scaling actions. It can be treated as a dynamic configuration catalogue for a Scalarm installation.

The internal architecture of the Information Manager is depicted in Fig. 5.6. It exposes two interfaces for external services: one for (de)registering services withing a Scalarm implementation and another for downloading source packages with Scalarm services. The former interface provides location information for each self-scalable service as well as actual locations of every instance of the service in the platform.

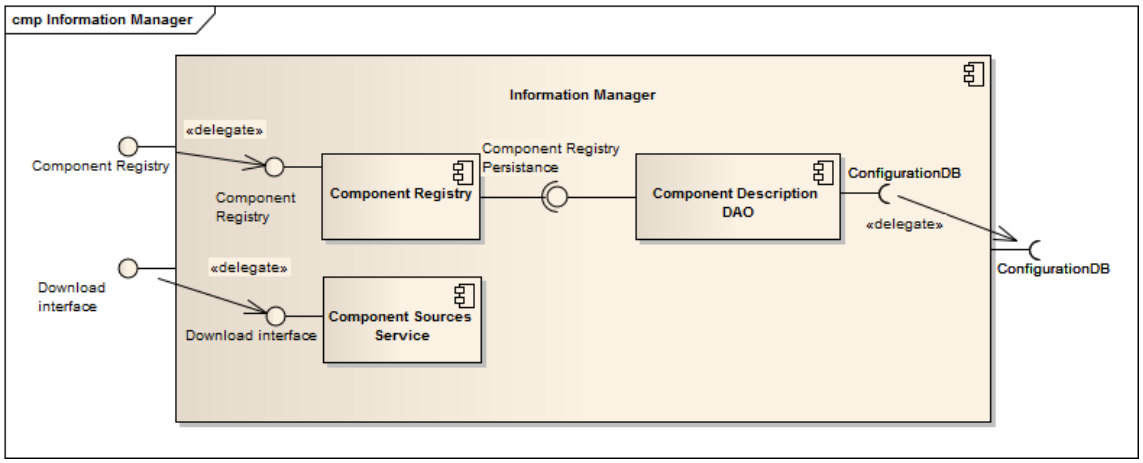

Figure 5.6: Internal architecture of the Information Manager.

This information can be used to determine where new instances should be started or which instances can be stopped. Information about each registered service is stored in a relational database.

5.2.5 Node Manager

To abstract access to computational resources, Scalarm provides a dedicated component called the Node Manager. It is a utility component which facilitate installation and management of other Scalarm services on computational resources. In addition, the Node Manager contains sensors to monitor system-level metrics, e.g. CPU and HDD utilization, along with application-level metrics such as service response time. Application-level metrics are gathered from instance log files. When a new component instance is started, the Node Manager observes and parses its log file to extract information about handled requests. This information is then sent to a dedicated monitoring system for further analysis.

Following installation, the Node Manager registers itself with the Information Manager to become visible to other components. It provides a RESTful HTTP interface for installation and management, i.e. starting, stopping and querying the status of Experiment, Storage and Simulation Managers.

5.2.6 Monitoring

Reacting to workload changes in a self-scalable service requires information about the current workload of each service instance. This information represents basic knowledge which can be used to adapt the self-scalable service to changes in its environment. Depending on the service purpose, monitoring can involve different

metrics, e.g. service response time (when considering a web service) or memory consumption (when dealing with a numerical algorithm). Even the process of collecting monitoring information can be a challenging task, especially in the context of a distributed system where different instances reside on geographically distributed servers.

To collect monitoring information each self-scalable service includes a dedicated module, called the Monitoring, which constitutes a distributed monitoring system. It consists of two separate elements: sensors, which periodically report monitoring data, and a central database service which stores this data. Sensors are built directly into Node Managers to couple the monitoring functionality with the deployment layer of Scalarm. As Node Managers handle installation of each Scalarm service, monitoring can only be enabled for active services. Even if there is no Scalarm service running, sensors monitor the workload of the underlying operating system. Monitoring data includes:

- basic system metrics, e.g. CPU utilization and free RAM,
- storage-related metrics, e.g. average length of HDD I/O queues, average time (in ms) for each I/O request to be served, amount of data read from and written to HDD, etc.,
- specific information about Scalarm service instances, e.g. response time for various requests.

Each Node Manager periodically dispatches this information to a central service, which stores it in a database. Collecting all information in a single database enables statistical analyses the self-scalable service workload which can identify performance and scalability bottlenecks. Furthermore, long-term historical data can be used to discover load patterns and even predict when new service instances should be started. Current version of the Monitoring Manager processes this information to be accessible remotely by other services (in particular the Scalability Manager) in a convenient way, i.e. as a response to scaling rule checks.

5.2.7 Scalability Manager

The end-user features of Scalarm are centered upon management of data farming experiments. In order to effectively provide such functionality, self-scalability is necessary. Unlike other existing systems, Scalarm addresses the problem of scaling both the master and worker parts of the platform. The self-scalability feature is provided in Scalarm by self-scalable services. Two elements constituting the Scalarm master, i.e. Experiment and Storage Managers, exploit this concept. While Simulation

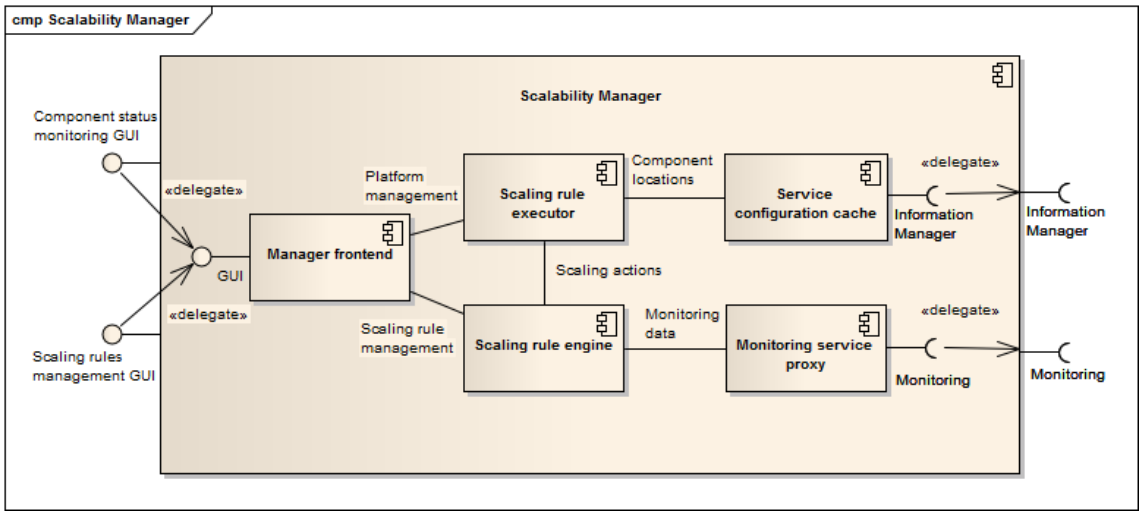

Figure 5.7: Internal architecture of the Scalability Manager.

Managers can be added by users at any time, e.g. to boost a given experiment, master elements automatically react to changing workload via scaling actions.

The module responsible for automatic MoE management of self-scalable services is called the *Scalability Manager*. This service is not visible to analysts who use the platform. Instead, it is configured and utilized by the administrator of a Scalarm installation. MoE management functionality involves:

- monitoring the status of service instances running on Node Managers registered in the Information Manager,
- handling scaling rule definitions, monitoring and enforcement,
- starting new instances and stopping existing instances of the supervised service.

The internal structure of the Scalability Manager is depicted in Fig. 5.7. It exposes a Graphical User Interface for manual monitoring and managing services supervised by Node Managers registered in the Scalarm Information Manager. In addition, a GUI for defining and managing scaling rules in the context of a specific self-scalable service is provided. Each scaling rule carries a condition which specifies when the corresponding scaling action should be performed. Conditions are monitored by the scaling rule engine using monitoring data from the Monitoring. If the current service workload satisfies a given condition, the relevant scaling operation is executed by the Scalability Manager, e.g. by deploying additional service instances.

Each scaling operation involves communication with the Scalarm Information Service to retrieve a list of resources supervised by Node Managers. In the context of starting or stopping component instances, remote Node Managers are utilized.

5.2.8 Load balancer

Load balancer is another module used by self-scalable services. It provides a single entry point to all instances of a given service. It receives each incoming request, forwards it to a service instance and returns the result to the user. It can be used to increase the performance and availability of a self-scalable service. In the case of stateless services the load balancer serves as a shared entry point for a cluster of service instances. In addition, the load balancer monitors the availability of each instance to ensure that no requests are forwarded to unavailable instances.

5.2.9 Cache

The *Cache* service manages shared memory for every instance of a self-scalable service. It can be used to persist the state of a service instance and to implicitly communicate with other instances which have access to this memory. Furthermore, cache memory can store data structures which rarely change. For example, when considering a database backend, cache can minimize the number of requests which actually need to be executed by the database management system.

5.3 Architectural Elements Supporting Scalability

Achieving massive scalability requires minimization of work that has to be processed sequentially. This is a difficult engineering challenge, especially when considering platforms that involve different technologies and components, e.g. web services and databases. A throughout analysis of existing software is required to identify actual and potential scalability bottlenecks. Such analysis usually starts at the architectural level, but needs to acknowledge implementation-related issues as well. Scalarm utilizes various mechanisms in each component to facilitate parallelization and increase performance. It should be noted that different mechanisms are utilized on the management and worker sides.

On the management side, Scalarm utilizes a non-relational database to store information about each simulation. When dealing with large-scale experiments, millions of simulations may need to be registered in the database. Efficient access to this data is one of the key determinants of Scalarm performance and scalability. Furthermore, the total amount of storage capacity required to handle dozens of experiments can easily exceeds the capabilities of a single machine. In addition,

operating a centralized database can result in poor performance when dealing with thousands of concurrent client connections. To eliminate this problem, Scalarm utilizes the concept of data sharding, which enables data distribution among multiple database instances in a transparent way. Each table from a sharded database has a designated partition key which specifies how to distribute its rows. In the case of simple queries predicated on the partition key, only those instances which actually hold the relevant part of the table are queried. In other cases, each read operation is executed in parallel on every database instance and partial results are then combined into the final result to maintain stable access time regardless of the number of database instances involved. To increase performance and scalability even further, data sharding has been extended to include Experiment Managers. In the context of a single experiment, each Experiment Manager is responsible for handling only a subspace of the parameter space. This approach minimizes the time necessary to decide which combination of input parameters' values should be executed next. Furthermore, this approach delays the insertion of a row describing the simulation and hence minimizes the required storage capacity. Although data sharding is handled automatically by the non-relational database management system used by Scalarm, i.e. MongoDB, the platform nevertheless remains responsible for database cluster management, i.e. adding or removing database instances when necessary. The distribution of read and write operations is transparent from the client's point of view as all database instances are grouped in a self-scalable service behind a single access point.

Another mechanism which supports scalability on the management side is data caching. Most use cases are divided into a number of steps, each of which is implemented as a separate request-response loop between platform components. When dealing with multiple instances of a single component, each request can be handled by a different instance. Even though static experiment-related data does not change throughout the course of the experiment, utilizing a number of stateless instances means such data may be retrieved from the database multiple times. To mitigate this issue each self-scalable service has access to a Cache module, which minimizes the number of database queries. This is especially useful when the experiment involves long access/computations on static data or rapid changes which are relevant to the user, e.g. updating the experiment progress bar which aggregates the states of every simulation in an experiment.

Running large-scale data farming experiments often requires extensive computational resources such as physical servers, virtual machines in the Cloud or Grid jobs. Scalarm provides basic management functionality for computational resources, i.e. it can allocate or release resources in a way specific for a particular computating infrastructure. Each computational resource is responsible only for starting a Simulation Manager instance, which then handles all the work related to execution of

actual simulations. Most existing task management systems treat workers as passive elements, i.e. units which are told to perform a certain task. This approach constraints scalability because all workers needs to be registered and accessible by the master. Nowadays this is hardly ever the case due to restricted network traffic and shortage of public IP addresses. Similar concerns arise in the context of Scalarm, which intends to enable execution of simulations on any computational resources, regardless of whether they are part of the master's network. Fulfilling this criterion would make it possible to combine resources from multiple administrative domains in a transparent way. Hence, Simulation Managers are implemented as active elements, i.e. they *pull* work to do from Experiment Managers, which are publicly accessible through a single entry point. This strategy allows Experiment Managers to know nothing about workers which run actual simulations. They can be managed by Scalarm, by the user (manually), or by any other custom client. As a result, Scalarm can handle a much larger pool of workers and computational resources than can be provided by a single private network.

5.4 Automatic Scalability Management

Traditional virtual platforms were designed to handle peak workloads. This usually resulted in poor resource utilization, as peak conditions occur very rarely. On the other hand, allocating insufficient resources could degrade performance when a large number of clients attempt to access the given platform simultaneously.

In Scalarm, this problem is mitigated by implementing self-scalability, i.e. efficient and comprehensive scalability management in an automatic way. To increase flexibility, the self-scalability feature is implemented on the level of self-scalable services, i.e. each such service is capable to scale itself independently. This feature comprises three separate elements:

- scaling rules which expresses expert knowledge about how and when the service should be scaled,
- an enforcement engine, implemented in Scalarm by the Scalability Manager, which monitors and executes scaling rules when necessary,
- a monitoring system, which provides information about the current load of each computational resource supervised by the service.

Scaling rules are introduced in this thesis to represent conditions and actions associated with software platform scalability. They formalize scalability management so that it can be performed by dedicated software in an automatic manner.

In the context of Scalarm scaling rules describe scalability management of each self-scalable service. Each service may have different resource requirements, hence its scaling rules should be based on different performance metrics. Moreover, scalability requirements can depend on specific simulations which are executed by Scalarm, e.g. simulations with large output sets will require more throughput from Storage Managers, while running many short simulations may increase the communication load between Experiment and Simulation Managers. Thus, scaling rules should be defined by administrators based on their knowledge regarding the computational infrastructure, combined with observations of running simulations.

While scaling rules represent knowledge about service scalability in a formal manner, a separate module is necessary to process this knowledge. In Scalarm the module that manages and uses scaling rules to provide self-scalability is called the Scalability Manager. It provides a Graphical User Interface for administrators to define, manage, and monitor scaling rules. Each time a new scaling rule is defined, a process is started to enforce the rule by monitoring the specified conditions and executing scaling operations if necessary.

Information required to handle scaling rules is provided by a dedicated monitoring system which is divided into two elements. Sensors (located in Node Managers) collect all available monitoring data from computational resources and push it to a centralized database. This database is managed by the second element, i.e. the monitoring service, which aggregates the available data to efficiently respond to requests concerning scaling conditions.

A self-scalable service with modules responsible for handling scalability management is depicted in Fig. 5.8. This service can be treated as a wrapper for an actual service, which provides business features to external clients. Within a self-scalable service the actual service can be instantiated multiple times on different computational resources, which are included in a computational resource pool supervised by the self-scalable service. The load balancer, which constitutes a single access point to the service, is responsible for delegating requests from clients to different service instances. If a given computational resource does not host a service instance, it is released. Moreover, computational resources include monitoring sensors which periodically report current workload to the Monitoring.

The administrator defines scaling rules using the Scalability Manager to enable self-scalability. The Scalability Manager then initiates supervising processes to enforce each defined rule. A supervising process periodically collects monitoring data from the Monitoring, checks if the rule condition is met and if so, executes scaling actions. Scaling actions typically involve starting new instances on idle computational resources or stopping existing instances. To select resources upon which new instances should be started, or to stop existing instances, the Scalability Manager queries the Information Manager which stores information about resources and in-

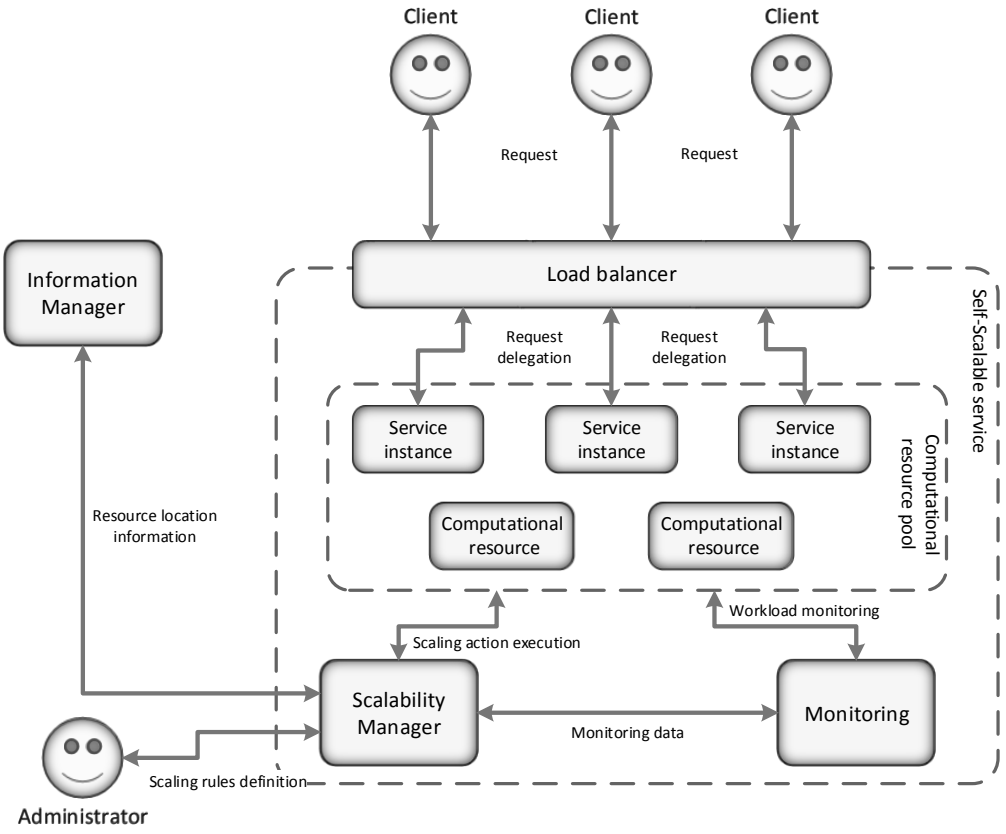

Figure 5.8: Overview of scalability management within self-scalable services.

stances of each scalable service.

Computational resources can be shared between self-scalable services. This feature is especially important when considering deployments with a fixed quantity of resources. In such cases self-scalable services can spawn new instances on resources which are already occupied by instances of other services – but only if the resource is underutilized. The current version of the platform does not implement a preemption feature which would enable service prioritization.

An important aspect of scalability management is ensuring that execution of a scaling rule will not trigger execution of other scaling rules as a result of the additional load caused by starting a new instance. To prevent this undesirable scenario, a so-called *cooldown* interval is enforced after each scaling action. During the cooldown period scaling rules are inactive and monitoring data from this period is not taken into account by Scalability Managers. Furthermore, when considering a Scalarm deployment with several self-scalable services which share computation

resources, a global cooldown period may become desirable. Such a feature would prevent one self-scalable service from interfering with others, although – on the other hand – local cooldown periods make services more independent. In the current implementation a global cooldown period was used, though it remains an area for further investigation as part of future work.

5.5 Implementation of Essential Use Cases

Scalability and performance, which are arguably the most important non-functional features of the Scalarm platform, depend heavily on implementation details of the most frequently executed algorithms. By "algorithms" we mean any action performed by the platform, from creation of experiments, through scheduling and execution of simulations all the way to aggregation and analysis of results. The architecture of a distributed system can support non-functional requirements by incorporating useful design patterns and good practices, though the actual implementation of basic use cases determines the final scalability and performance levels. Achieving the specified non-functional features requires that two key conditions be met:

1. Execution of algorithms for different clients should be as independent as possible.

2. The algorithms' execution time should not depend on problem size.

Meeting the first condition increases the speedup resulting from allocation of new resources, while fulfilling the second condition prevents performance degradation when the problem size increases. In addition, the system should be able to utilize any addition resources provided to it.

In practice, meeting both conditions is nearly impossible due to complexity of software. Even common web applications involve multiple technologies, e.g. databases, web servers and messaging libraries, each of which has to be taken into account when considering scalability. In the following subsections the author describes the implementation of basic use cases in more detail, focusing on the scalability context. The description covers creating new data farming experiments, executing simulations and extending data farming experiments. These use cases were selected as a representative set of algorithms which are most frequently executed by Scalarm.

5.5.1 "Creating a data farming experiment" use case

Each new data farming experiment begins with questions about the studied phenomena, which should be answered by the experiment. Hence, the use case concerning

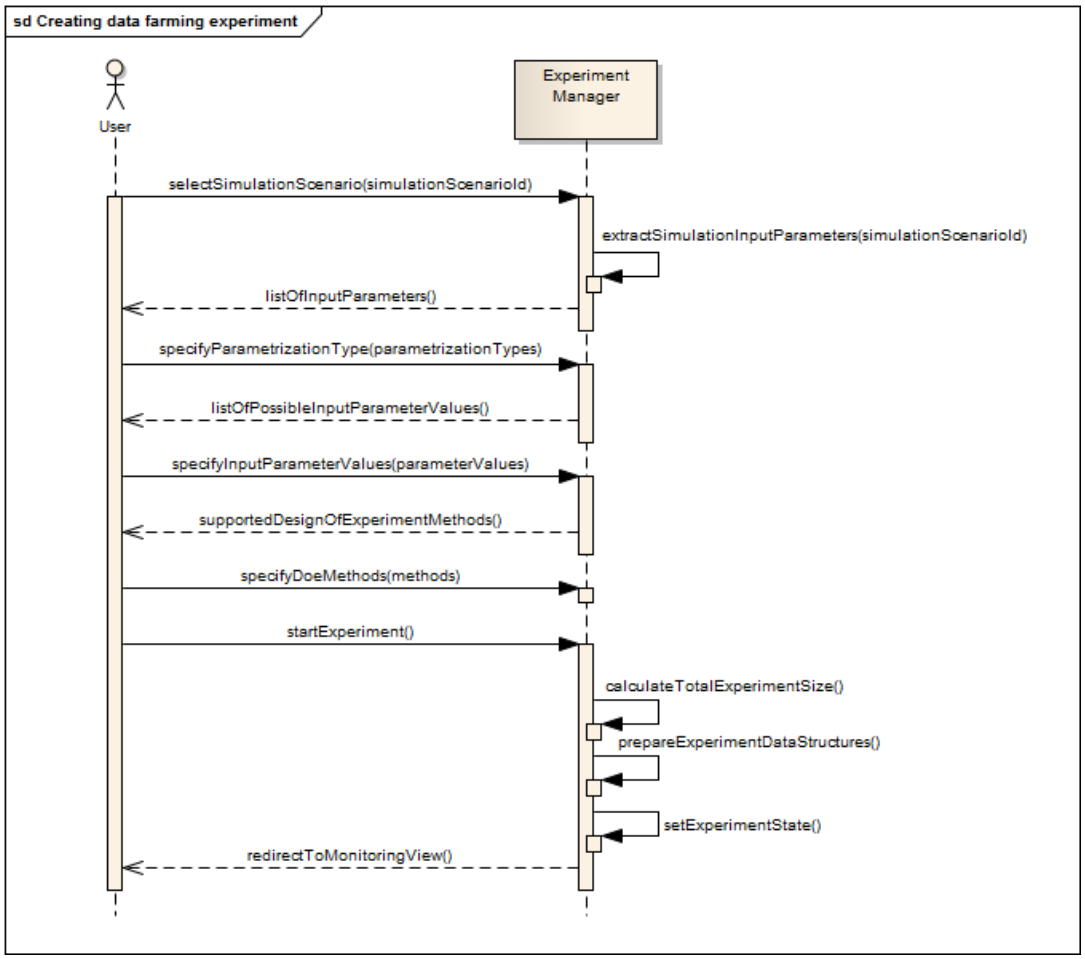

Figure 5.9: Sequence diagram of the "Creating data farming experiment" use case.

creation of a new data farming experiment involves preparation of the simulation input parameter space. Scalarm intends to facilitate this phase by organizing the user's workflow as depicted in Fig. 5.9. This use case involves two actors, namely the analyst and the Experiment Manager component.

The first step is to select the simulation scenario which will be executed during the experiment. The Experiment Manager extracts and returns information about input parameters from the simulation's description along with possible parametrization types, e.g. random values, ranges of values, etc. The analyst then selects parametrization types and provides the required arguments specific for each parametrization type, e.g. in the case of a range of values, boundaries and step values would need to be provided. This information enables Scalarm to calculate the initial size of the experiment. However, to increase efficiency, the analyst can group input parameters and apply DoE methods to minimize the number of simulations

required in order to obtain meaningful information about the simulated phenomena. Following this step the analyst can start the experiment, which results in preparing the necessary data structures on the Experiment Manager's side.

Each step of the presented use case involves a separate request-response loop. Hence, each step can be handled by a different Experiment Manager instance as long as the state of the use case is passed in requests. As each experiment has a unique id, multiple experiments can be created by the platform simultaneously. The state of each use case is partially stored in a database (experiment-related information) and partially in the invoked request-response operations (information provided by the user). It is worth noting that the actual input parameter space is not generated during this step. This is a conscious design decision. Large-scale experiments require extensive storage capacity as they involve millions of simulations, each of which has to be described separately to enable further analysis. By generating the input parameter space on demand, Scalarm minimizes storage space requirements and avoids the large computational cost of upfront generation.

A description of this use case in the context of the EDA EUSAS project is presented in Chapter 7.

5.5.2 "Simulation execution" use case

Once a new data farming experiment has been started, actual simulations can be run with input parameters selected from the previously defined parameter space. The simulation execution use case describes how each simulation is scheduled from the Simulation Manager's perspective. This use case is performed for each simulation and therefore has great impact on the scalability and performance of Scalarm. Furthermore, this use case can be performed many times in a short period of time if many Simulation Managers are present in the system. It calls for cooperation between all main functional components of Scalarm: Simulation, Experiment and Storage Managers. The platform's end users (analysts) are not directly involved because once an experiment begins, Scalarm handles simulation scheduling automatically.

The sequence diagram for this use case is presented in Fig. 5.10. The initial state of the platform includes at least one already running experiment and a new Simulation Manager started on a computational resource. The first step performed by the Simulation Manager is to check whether any experiment is currently running. If so, the Simulation Manager downloads simulation code and then repeatedly requests information about successive simulations which should be executed. Each request for simulation input values requires communication between Experiment and Storage Managers to obtain a simulation description using the experiment scheduling policy (which is random by default).

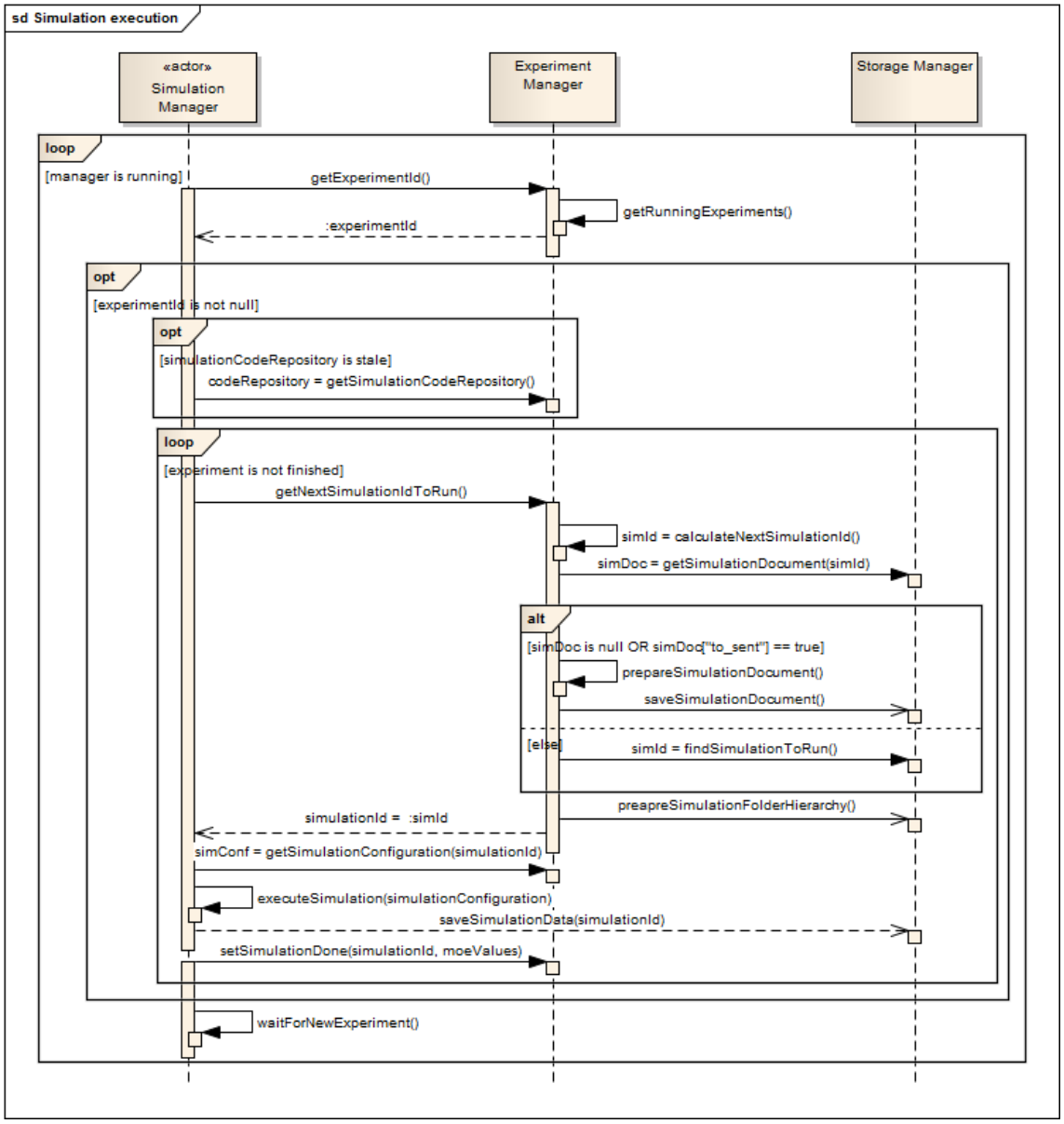

Figure 5.10: Sequence diagram of the "Simulation execution" use case.

Within a single experiment the simulation parameter space is partitioned between the available Experiment Managers, i.e. each Experiment Manager schedules simulations based on its unique id and the total number of available Managers.

Preparing each simulation is therefore independent of the size of the parameter space. This approach can lead to collisions between Experiment Managers if new instances of the Experiment Manager are added to an already running experiment. As a result, additional collision detection functionality is added to this step.

Once the simulation concludes, the Simulation Manager uploads its textual/binary outcome, e.g. log files, directly to Storage Managers, while information about Measure of Effectiveness (MoE) values are sent to the Experiment Manager along with a notification of simulation completion. MoE values are key indicators of simulation results. This information represents basic knowledge about the simulated phenomena and should be processed by Experiment Managers. However, simulations can also produce additional binary/text data, which can be useful for in-depth studies. This data is uploaded directly to Storage Managers as it is not used in analyses performed by Experiment Managers.

Similarly to the previously described use case, each request to Experiment or Storage Managers can be handled by a different instance. Communication between different service types is governed by the self-scalable service abstraction, with a load balancer distributing requests to different service instances depending on their current load. To increase scalability of the Scalarm platform, there is no list of running Simulation Managers on the Experiment Managers' side. Instead, Simulation Managers are active workers and use the "pull" approach to obtain input values for simulations until the experiment input space is fully explored. Once this happens, the Simulation Manager can release any utilized computational resources. Following creation, static information about each simulation is cached by Experiment Managers in a Cache component to minimize the number of database queries. In addition, static information, about the experiment for which the simulation is executed is also cached between requests. If there are no more simulations to run, the Simulation Manager exits and releases its assigned computational resources.

5.5.3 "Extending an experiment" use case

Having completed a number of simulations, statistical analysis can be performed on the available partial results. Such an approach is suitable for exploratory experiments, i.e. whenever the user does not possess in-depth understanding of the studied phenomena beforehand. Typically, a large, sparsely sampled experiment space can be initially selected. Statistical analysis is then conducted to discover correlation between simulation input parameters and output MoE. Based on this analysis, additional input values can be added to the experiment parameter space, e.g. to analyze a previously omitted subspace in detail. The parameter space expansion process is

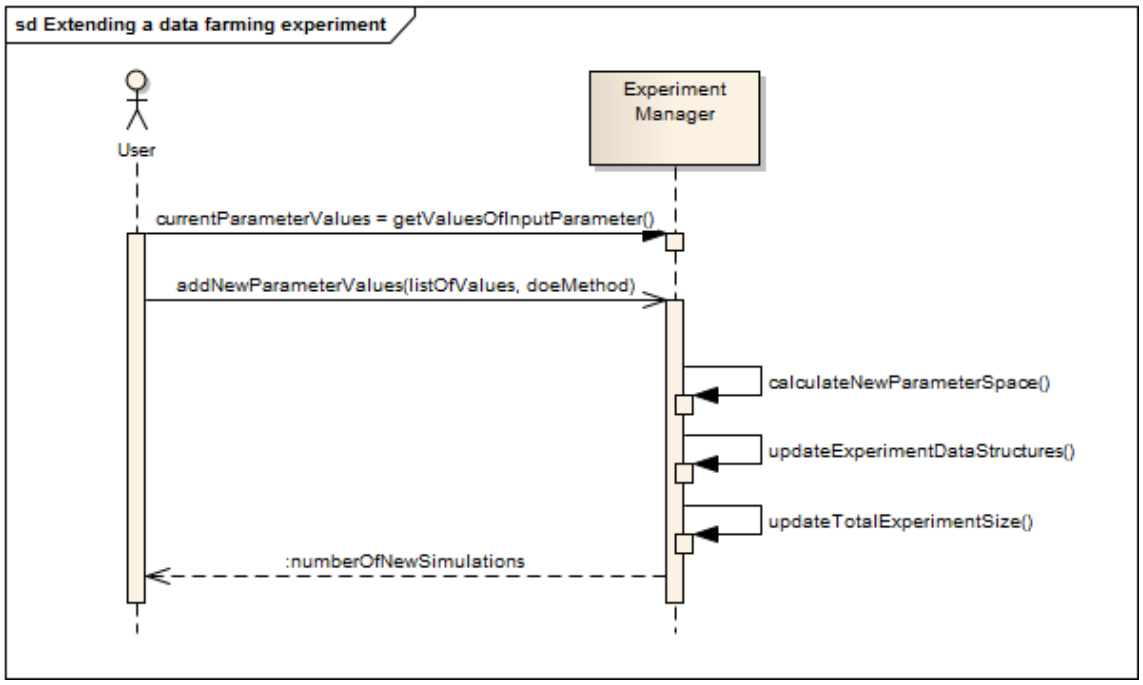

Figure 5.11: Sequence diagram of the "Extending a data farming experiment" use case.

depicted in Fig. 5.11.

The current version of Scalarm supports extending experiments in a piecemeal fashion, with regard to a single input parameter at a time. The analyst first specifies new values for selected input parameters. Based on this information, Scalarm modifies the data structures which describe the experiment and returns the new size of the experiment's parameter space. As the expansion process alters the experiment's basic properties, all cached data about the experiment needs to be marked as out of date (note, however, that the shared memory paradigm renders cache invalidation operations straightforward). Additionally, there is now a risk of using invalid information since the parameter space can only be expanded and not reduced.

Chapter 6

Experimental Evaluation

In this chapter the author presents experimental evaluation of the Scalarm platform. The evaluation is divided into three parts. The first part concerns support for heterogeneous computational infrastructures, i.e. private servers, Grids and commercial Cloud sites. The second evaluation test focuses on massive scalability of the platform in terms of allocating varying amounts of computational resources to the management part of the platform while conducting experiments of various sizes. Based on the obtained results, several scalability-related metrics are calculated and discussed. The final part concerns the self-scalability feature, i.e. the platform's ability to execute scaling actions automatically, based on predefined scaling rules and workload-related information. Results from the second set of sets provide a starting point for analysis of the self-scalability feature and its advantages, especially in terms of performance improvements and maintenance costs.

6.1 Evaluation Objectives

The main goal of the Scalarm platform is to handle large-scale data farming experiments. Throughout this dissertation the author has identified two key non-functional requirements which have to be met in order to achieve this goal, namely *massive scalability* and *self-scalability*. As previously mentioned, Scalarm intends to fulfill both requirements using the concepts introduced in this dissertation: *self-scalable services* and *scaling rules*. This experimental evaluation was conducted to verify both aspects.

Besides non-functional requirements, an essential aspect of Scalarm is its functionality in terms of supporting the data farming methodology. In particular, supporting heterogeneous computational infrastructures is crucial in the context of large-scale data farming experiments. However, it is difficult to validate data farming-related features without introducing a real-life use case. Chapter 7 describes such a use case, which involves training of security forces.

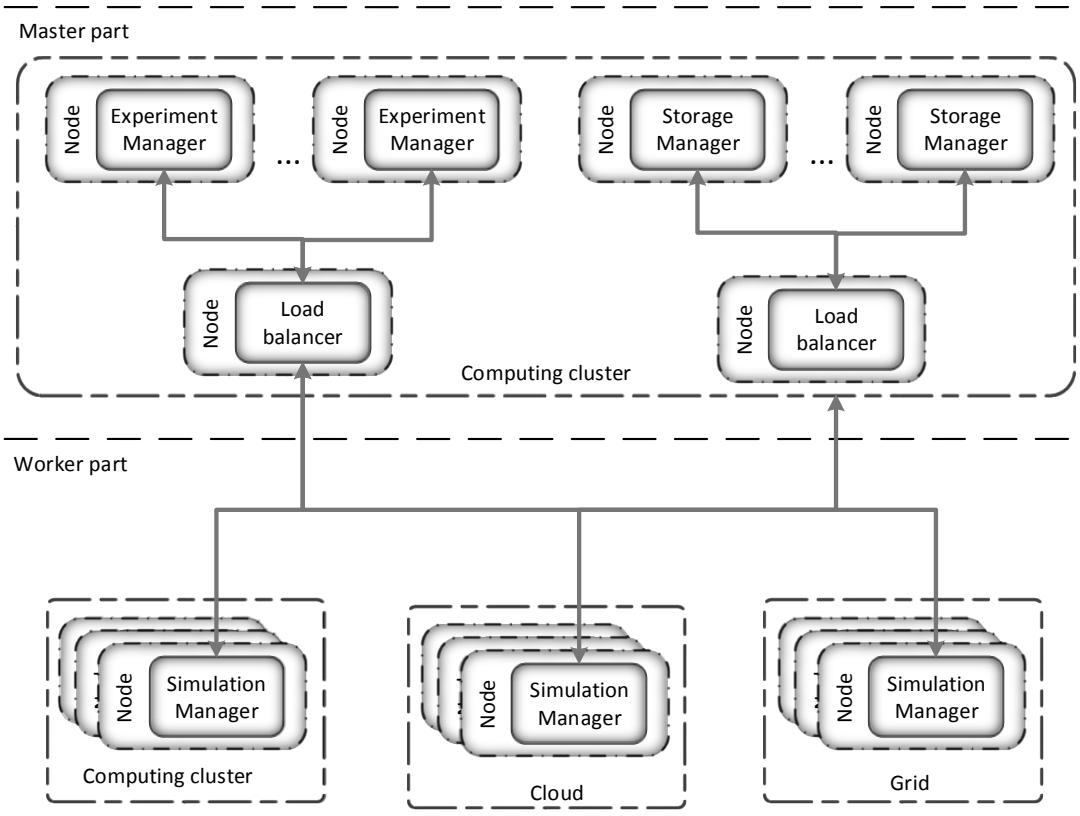

Figure 6.1: Testing environment for evaluation of massive self-scalability.

6.2 Evaluation of Massive Scalability

When considering a "master-worker"-based system, massive scalability refers to both parts of the platform (Fig. 6.1):

- Scaling out the "worker" part, i.e. Simulation Managers, which are independent from one another, seems trivial: one simply needs to start a new instance of a Simulation Manager on a separate server. There are no constraints on the number of running Simulation Manager instances other than those related to the amount of available computational resources.
- Scaling the "master" part, i.e. Experiment and Storage Managers, is a more challenging task due to the need to share state, i.e. information about data farming experiments.

This evaluation will address the problem of scaling the "master" part only, which can be described as the ability to increase *throughput* as the number of Experiment and Storage Manager instances grows. The *throughput* of the platform can be defined as the number of simulations scheduled in a given interval. To measure this effect the author uses metrics introduced in Chapter 4, along with the throughput metric.

6.2.1 Testing scenario

The baseline scenario for the presented evaluation involves executing a “foo” data farming experiment, i.e. an experiment, which runs a simulation that does not actually perform any calculations, using a predefined resource configuration. Executing the “foo” data farming experiment implies executing the simulation for each element of the experiment's input space. The goal of this evaluation is to measure the efficiency of experiment management by measuring the scalability of the platform's "master" part.

A single test case comprises execution of an individual foo data farming experiment with a predetermined input space, using a well-defined resource configuration. The key metric measured during scalability tests was *experiment execution time*, which can be defined as *time elapsed between requesting execution of an experiment based on the provided input space specification and the final simulation being reported as complete*. The simulation scheduling policy is random, i.e. each element from the input space can be randomly selected with a uniform distribution of probability.

The described testing scenario has two parameters:

- *experiment size*, i.e. the number of elements included in the experiment input space. For each input space element a single foo simulation was executed.
- *resource configuration*, which denotes the number of servers used to run Experiment and Storage Managers, i.e. the "master" part of the platform. For brevity's sake each resource configuration label is represented as the following pair: (<experiment manager count>, <storage manager count>).

The final point to consider in such tests is the number of "workers", i.e. Simulation Managers, running in parallel. In our scenario we intended to achieve the highest possible throughput and hence needed to start enough Simulation Managers that each additional instance would actually decrease throughput by overloading the platform. This count of Simulation Manager instances was empirically pegged at 25 per each Experiment Manager instance.

For the purpose of the presented evaluation the following experiment size values were tested:

- 100 000

- 200 000
- 500 000
- 1 000 000
- 2 000 000
- 5 000 000

This range should satisfy the requirements of most users while enabling the author to demonstrate the full capabilities of the platform. Each experiment was executed twice.

Regarding the second parameter, i.e. resource configuration, the options listed in Table 6.1 were used for experimental evaluation.

Table 6.1: Resource configurations tested during experimental evaluation.

Resource configuration			
Experiment Managers	Storage Managers	Simulation Managers	Configuration label
1	1	25	Configuration(1, 1)
2	2	50	Configuration(2, 2)
4	4	100	Configuration(4, 4)
8	8	200	Configuration(8, 8)

6.2.2 Testing environment

The properties of the testing environment are crucially important in any experimental evaluation. In our case the environment has to imitate a production environment, i.e. an environment where large-scale experiments would be actually conducted, to enable us to determine real-life scalability and performance features of the platform. The author decided to use a computing cluster, along with a Grid environment provided by the PLGrid Plus project [49], to run platform tests. The architecture of the testing environment, depicted in Fig. 6.1, conforms to the architecture of Scalarm where each component is executed on a separate physical server.

For the purposes of the presented study, Experiment and Storage Managers were run on standard nodes connected through a 10 GbE network switch. Each worker node had a 1 GbE link to the switch and shared the following parameters:

- CPU: 2x Intel Xeon CPU L5420 @ 2.50GHz (4 cores each)
- Memory: 16 GB RAM
- Disk: 120 GB hard drive (5400 RPM)
- Operating system: Ubuntu Linux 10.04.1 LTS

6.2.3 Scalability evaluation results

The execution time (in seconds) of each scalability test is listed in Table 6.2. During evaluation, a separate data farming experiment was created for each experiment size and resource configuration. The value listed in the configuration column indicates the measured execution time. The size of the experiment is listed in the leftmost column of the table.

Several issues need to be pointed out here. First of all, the more resources Scalarm has available, the better its performance. In almost all cases the performance gain grows linearly with experiment size. Comparing configurations (1, 1) and (2, 2) reveals a performance gain of nearly 50%. For configurations (1, 1) and (4, 4) the corresponding gain is above 65%, while for configurations (1, 1) and (8, 8) it exceeds 80% (on average). This linear performance gain achieved while scaling up from 2 to 16 servers is the first positive confirmation of the platform's scalability.

Table 6.2: Execution time [s] for experiments of varying sizes, depending on the Scalarm resource configuration.

Execution time [s]				
Experiment size [#simulations]	Configuration (1,1)	Configuration (2,2)	Configuration (4,4)	Configuration (8,8)
100 000	1049	493	321	240
200 000	2219	1068	664	433
500 000	5553	3692	1880	1039
1 000 000	11499	7881	4006	2186
2 000 000	31032	15203	10614	4076
5 000 000	98553	64797	31272	13860

The second observation focuses on the execution time of increasingly larger experiments using a single configuration. Regardless of the configuration parameters, the growth in the execution time of experiments depending on the number of simulations is faster than linear. With configuration (1,1) the difference in execution time for experiments involving 100 000 and 200 000 simulations respectively is 111%, but for 1 000 000 and 2 000 000 simulations it works out to nearly 170%. This is caused by increasing simulation management overhead. Each simulation is represented in Scalarm by a row in a non-relational database, which is supervised by the Storage Manager (Fig. 5.4). The performance of such databases depends on the I/O subsystem, especially when concerning millions of rows. As the database volume grows larger (reaching millions of rows on a single host) data access operations tend to take longer than expected.

Speedup Based on the collected measurements, the scalability metrics introduced in Chapter 4, can be calculated. The first such metric, which provides basic infor-

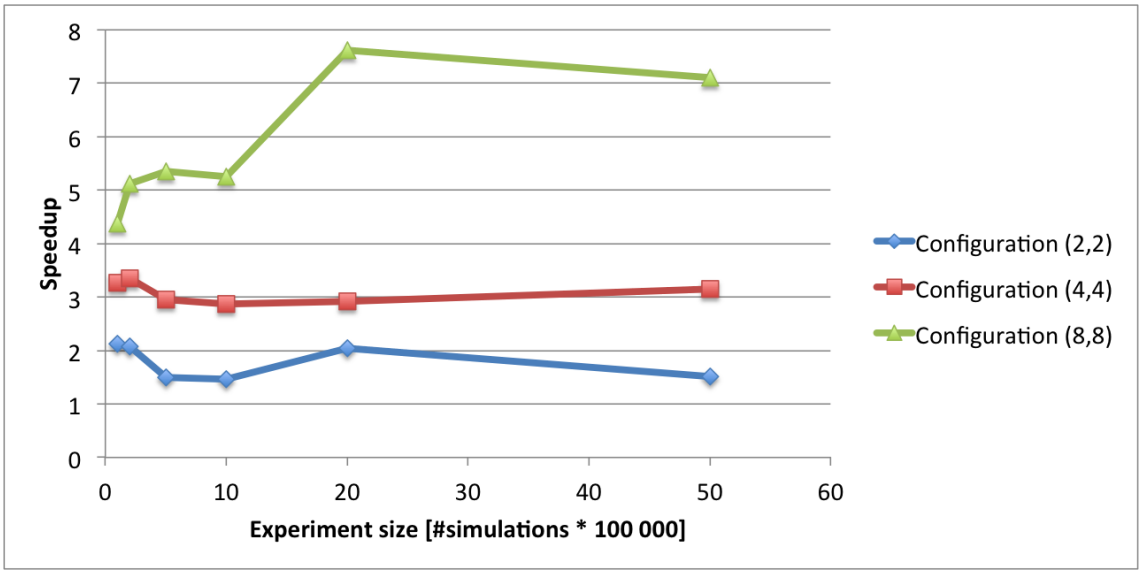

Figure 6.2: The speedup metric for different experiment sizes and resource configurations.

mation about the platform's scalability is speedup. We will use the basic formula presented in Eq. 4.1, where T_S is the execution time obtained with configuration (1, 1), i.e. with a single instance of the Experiment Manager and a single instance of the Storage Manager; and T_N is the execution time obtained with configuration(N, N), where N denotes the number of Experiment and Storage Manager instances respectively. The resulting speedup is depicted in Fig. 6.2. As already discussed, the highest possible speedup for configuration(N, N) is N, however the presented results indicate that actual speedup is lower in all cases, and that the difference between the measured and ideal speedup increases along with N, as listed in Table 6.3.

Table 6.3: Mean speedup values for various resource configurations.

Scalarm speedup		
Resource configuration	Mean speedup	Standard deviation
Configuration(2, 2)	1.78	0.29
Configuration(4, 4)	3.08	0.17
Configuration(8, 8)	5.80	1.04

For configuration (2, 2) and configuration (4, 4) the calculated speedup is quite stable, with standard deviation of 0.29 and 0.17 respectively. Speedup curves are relatively similar, with only one exception, i.e. speedup for configuration (4, 4) and experiment size 2 000 000 is lower than expected. The experiment execution time for this resource configuration was relatively high compared to other resource configurations. This might be caused by using a production Grid infrastructure to

run Simulation Managers, with difficult-to-predict perturbations in job scheduling time.

Interestingly, configuration (8, 8) provided greater speedup for larger experiments than for smaller ones: for experiments smaller than 2 000 000 simulations the average speedup was 5.025, while for larger experiments the corresponding value was 7.362. This can be explained by looking at execution times, which are more than 8 times greater for large experiments than for smaller ones. Running such massively scalable experiments makes it difficult to start all Simulation Managers at the exact same moment. The utilized Grid environment generated certain delays when scheduling large numbers of Simulation Managers. Unfortunately, since we used a production infrastructure, shared with a large number of users, this factor is unpredictable. To mitigate this problem, scheduling of Simulation Managers began before the experiment actually commenced (although note that an overly large initialization delay will cause idle Simulation Managers to stop).

In the case of configuration (8, 8), starting all 200 Simulation Managers (Table 6.1) required to saturate the platform took longer than the actual test for experiments with fewer than 2 000 000 simulations. Hence, allocating this many "master" servers did not result in the expected speedup. For larger experiments all scheduled Simulation Managers were started and the experiment was conducted much faster compared to configuration (1, 1). As a result, the average speedup of configuration (8, 8) exceeds 7.1, which is close to the ideal value of 8.

Efficiency The second commonly used scalability-related metric is efficiency. It denotes how the platform copes with additional resources in terms of the utilization level. Aggregated values of this metric are shown in Fig. 6.3. The ideal value is 1, regardless of the amount of resources in a given configuration. Scalarm's efficiency is 0.8 on average, and varies between configurations, especially for experiment sizes lower than 1 000 000 simulations. For larger experiments, efficiency is very similar and does not depend on the configuration. Configurations with fewer resources provide better efficiency for smaller experiments than configurations with more resources. This is expected, since additional resources require additional effort to balance the workload, which is profitable only for sufficiently large experiments. Moreover, the efficiency of configuration (2, 2) tends to decrease along with increasing experiment size, while the efficiency of configuration (8,8) exhibits the opposite tendency. This can also be explained by referring to the workload balance effort. Configuration (8, 8) requires much more demanding experiments (in terms of the number of simulations) to provide high throughput. The efficiency of Configuration (4,4) is the most stable, with an average value of 0.77 and a standard deviation of 0.049.

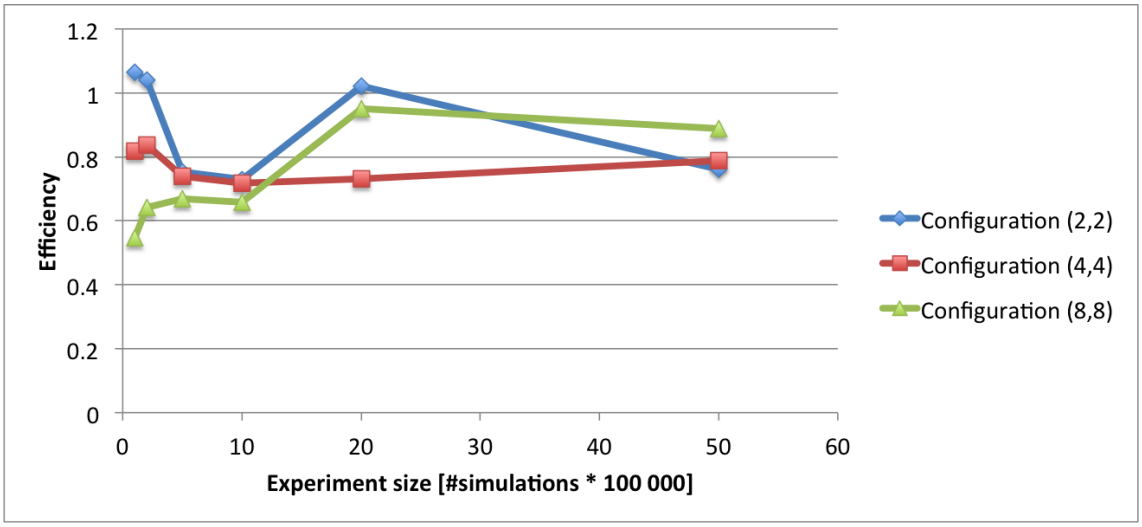

Figure 6.3: Efficiency of Scalarm for different experiment sizes and resource configurations.

Efficiency-based scalability The efficiency metric can be used to calculate platform scalability between scales N_1 and N_2, as described by Eq. 4.8. In our case, platform scale is measured by the number of servers running the "master" part. Four different platform scales can be distinguished here, each twice the size of the preceding one. However, as we use efficiency to calculate scalability, only two transitions can be defined (between 4 and 8 servers and between 8 and 16 servers).

Efficiency-based scalability can be divided into the following categories:

- "S_E" lower than 1 means that the platform loses efficiency when its scale increases. This is the most common case in real-life applications.
- "S_E" of exactly 1 means that transition between scales does not influence efficiency at all and that the platform can therefore be scaled out indefinitely.
- "S_E" greater than 1 indicates that efficiency grows along with scale, i.e. the platform becomes more efficient at larger scales.

According to the presented definition, the scalability of a truly scalable platform is equal to or greater than 1 regardless of the problem size and scale. However, this is hardly ever the case – in practice a platform is considered scalable when the value of the scalability metric is greater than some predefined threshold, e.g. 0.7 or 0.8. This means that the platform actually loses efficiency when scale increases, but that the efficiency falloff curve is shallow enough to make the platform efficient even when dealing with large problems.

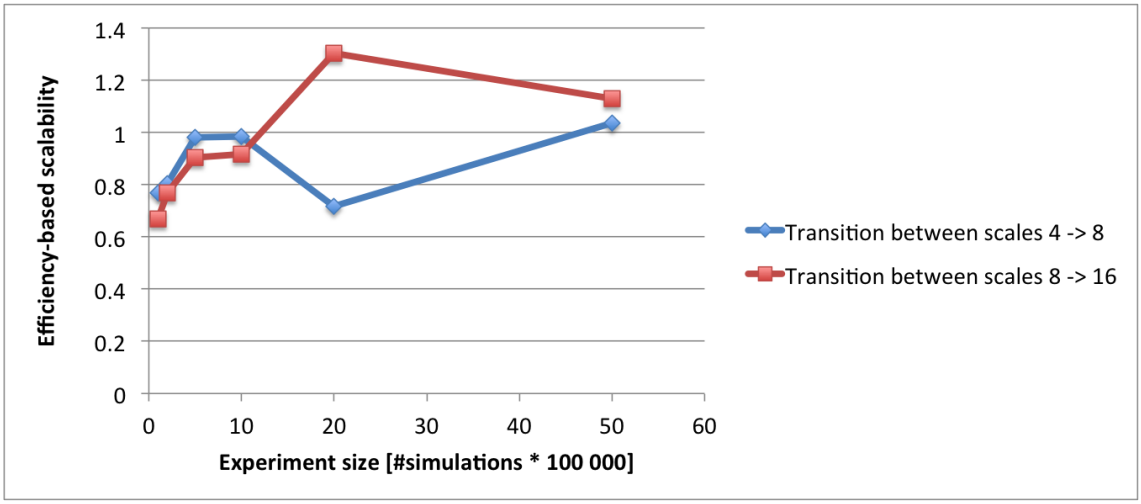

Figure 6.4: Efficiency-based scalability for different experiment sizes.

In the case of Scalarm, the author calculated the efficiency-based scalability for two scale transitions, namely from 4 to 8 servers and from 8 to 16 servers. Collected values are listed in Fig. 6.4. In most cases Scalarm provides scalability close to or greater than 0.8, with an average value of 0.91, which is better than expected in a real-world system. Interestingly, efficiency-based scalability tends to grow with experiment size, especially at larger scales. This means that additional resources are efficiently utilized when necessary, i.e. when the experiment is too large to be processed efficiently by a small-scale configuration.

Productivity While efficiency-based scalability provides information on how the platform loses efficiency along with increases in scale, it does not include cost- and quality-related information. This information has been taken into account when formulating the productivity concept, which is expressed by Eq. 4.9. To calculate platform productivity at scale N, we need to provide information about:

- throughput,
- average quality value of each response,
- cost of maintaining the platform at scale N.

Similarly to efficiency-based scalability, the scale of Scalarm is determined by measuring the aggregate number of servers used to run Experiment and Storage Managers.

Throughput Information about Scalarm throughput, i.e. the number of simulations scheduled in a given interval, is presented in Table 6.4. Scalarm provides very high throughput compared to common schedulers used in production Grids, e.g. Condor or PBS, owing to the Pilot jobs concepts, i.e. acquiring computational resources with Simulation Managers and then scheduling simulations directly using the *pull* mechanism. In [114], the authors measure the throughput of Condor and PBS tools by scheduling many short tasks. The measured throughput for Condor equals 11 [tasks/s], while for PBS it is less than 1 [task/s].

The throughput of Scalarm is similar or (in certain situations) higher than that offered by Falkon [114], which is optimized for efficient scheduling of small tasks. However, in contrast to Falkon, Scalarm is not constrained by the amount of resources dedicated to the master part of the platform. As a consequence, we are confident that Scalarm throughput is not limited to the values listed in Table 6.4.

Table 6.4: Scalarm throughput [simulations/second] for data farming experiments of varying sizes, depending on resource configuration.

Scalarm throughput [simulations/s]			
Experiment size [#simulations]	Configuration (2,2)	Configuration (4,4)	Configuration (8,8)
100 000	203	312	417
200 000	187	301	462
500 000	135	266	481
1 000 000	127	250	457
2 000 000	132	188	491
5 000 000	77	160	361

Response value The second factor in the productivity formula refers to the *response value*, i.e. the *average quality value of each response*. It is a vague and platform-specific coefficient which can quite difficult to determine, especially when we involve metrics related to platform response delay, availability or likelihood of timeouts. For the purposes of this evaluation we apply a formula similar to the one presented in [106], which considers the mean platform response time compared to a target value. In our case the response time of Scalarm reflects the total overhead of the platform when executing simulations for a single input space element, i.e. the time spent in the Experiment and Storage Managers. To determine the target value, let us decompose this overhead. In order to complete each simulation, three HTTP requests to Scalarm are required:

1. obtain a id,
2. download parameter values for the simulation,

3. upload results.

For the purpose of the presented evaluation, the author decided to relate the response target value metric to the aggregate response times for all three requests. Completing each request under low load conditions may take up to 10 ms. Hence, the response target value is taken as 30 ms. The value function is depicted in Eq. 6.1.

$$response\_value(N) = \frac{1}{1 + \frac{avg\_response\_time(N)}{response\_target\_value}} \tag{6.1}$$

where N denotes the platform scale while $avg\_response\_time(N)$ denotes the mean overhead of the platform when executing a simulation for a single input space element. The maximum $response\_value$ approaches 1 when $avg\_response\_time$ of the platform is close to 0. When $avg\_response\_time$ equals the $response\_target\_value$ (in our case, 30 ms), the $response\_value$ is $\frac{1}{2}$. The greater the $avg\_response\_time$ value, the smaller the corresponding $response\_value$. The proposed $response\_value$ function provides better differentiation of cases when $avg\_response\_time$ is close to the target value than for cases where $avg\_response\_time$ is much higher than the target value.

The measured Scalarm response values are collected in Table 6.5. As expected, the response value rises when more resources are added to the platform, and decreases along with increases in experiment size. However, the greater the scale, the shallower the corresponding falloff in response values.

Table 6.5: The Scalarm response value metric depending on resource configuration.

Response value			
Experiment size [#simulations]	Configuration (2,2)	Configuration (4,4)	Configuration (8,8)
100 000	0.859	0.903	0.926
200 000	0.849	0.9	0.933
500 000	0.802	0.889	0.935
1 000 000	0.792	0.882	0.932
2 000 000	0.798	0.849	0.936
5 000 000	0.698	0.828	0.915

Cost The final component of the productivity formula is cost. It is difficult to estimate the real cost of the used infrastructure when working at an academic computer center. Hence, we have decided to apply the Amazon Cloud [32] price list and specifically, the "Double Extra large" instance selection (30 GB RAM, 8 virtual cores, high I/O performance), which costs $1.160 per Hour. However,

instead of paying for each computational hour, our cost estimation assumes per-second payment granularity. This assumption eliminates overhead when dealing with smaller experiments in a large-scale environment.

The calculated cost [$] for each evaluated experiment is listed in Table 6.6. For experiments smaller than 2 000 000 simulations, configuration (2, 2) appears to be the most cost-effective one, while for larger experiments configuration (8, 8) pulls ahead. Configuration (4, 4) is never the most cost-effective one but remains a compromise choice for all experiment sizes.

Table 6.6: Total cost [$] of executed tests, estimated using the Amazon EC2 price list.

Estimated cost of running the platform [$]			
Experiment size [#simulations]	Configuration (2,2)	Configuration (4,4)	Configuration (8,8)
100 000	0.635	0.827	1.237
200 000	1.377	1.712	2.232
500 000	4.759	4.846	5.357
1 000 000	10.158	10.327	11.27
2 000 000	19.595	27.361	21.014
5 000 000	83.516	80.612	71.456

Productivity Finally, the author calculated the productivity value (using Eq. 4.9) for different scales and experiment sizes (depicted in Fig 6.5). Regardless of the scale, productivity decreases exponentially with increasing experiment size. For smaller experiments, i.e. fewer than 500 000 simulations, the value of productivity is very similar for each tested scale, e.g. for experiments with 200 000 simulations, productivity of the platform at scale 16 is only 1.675 times greater than at scale 4. One the other hand, differences between productivity values for different scales are significant for large experiments, e.g. for an experiment with 2 000 000 simulations, productivity of the platform at scale 16 is 7.178 times greater than at scale 4.

Furthermore, size-dependent decreases in productivity are steeper at smaller scales, e.g. at scale 4, the productivity value drops by a factor of 4 for each successive experiment size, while at scale 16, the corresponding ratio is only 2.5. This is another confirmation that Scalarm can utilize large amounts of resources efficiently.

Productivity-based scalability While efficiency-based scalability expresses efficiency under changing scale conditions, productivity-based scalability expresses similar behavior of the productivity metric. By invoking the cost of physical resource usage, productivity-based scalability appears more suitable to business-oriented use

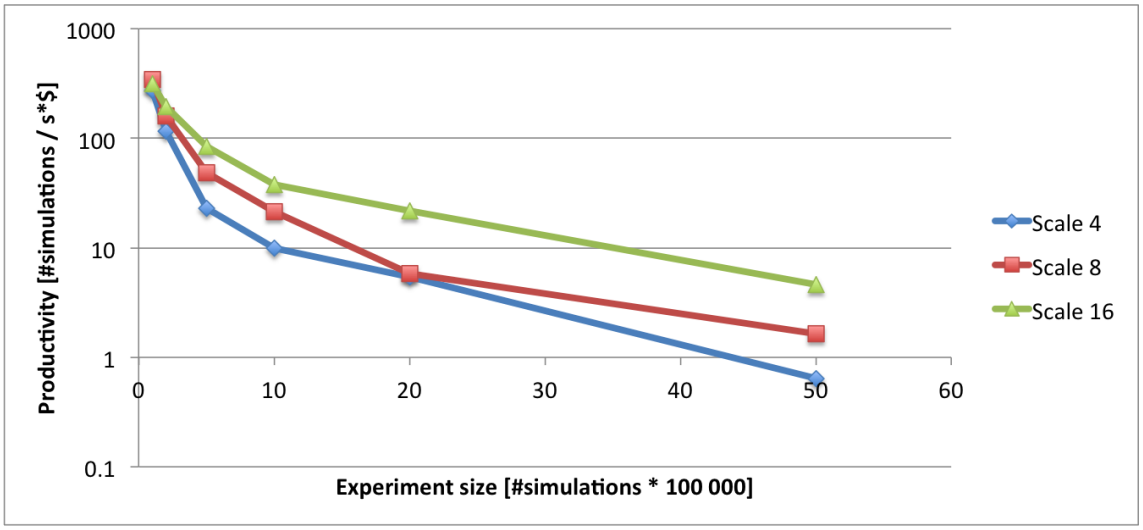

Figure 6.5: Scalarm productivity for different experiment sizes and scales.

cases. Since Scalarm can easily benefit from the elasticity of Cloud environments, this type of scalability is certainly important when considering real-life scenarios. The calculated values of productivity-based scalability for Scalarm are presented in Fig. 6.6. In most cases Scalarm provides scalability greater than 1 (with the average value being 1.89), which indicates superlinear scalability. In the context of productivity, this means that when upgrading from a smaller to a larger scale, productivity rises by almost 90% on average. In our tests scale was doubled for each subsequent experiment, i.e. the amount of servers used to run the master part of Scalarm increased by a factor of 2 each time. Furthermore, the value of productivity-based scalability tends to rise (with only two exceptions) along with the experiment size, which means that additional resources are efficiently utilized when necessary, i.e. when the experiment is too large to efficiently execute upon a small-scale configuration. The two exceptions relate to speedup values obtained for 2 000 000 simulations under different resource configurations (Fig. 6.2), as follows:

- for the 4 -> 8 scale transition, the "S_F" value decreases, which is related to the relatively small speedup between configuration (2, 2) and configuration (4, 4) for this experiment size,
- for the 8 -> 16 scale transition, the "S_F" value peaks, which is related to the relatively large speedup between configuration (4, 4) and configuration (8, 8) for this experiment size.

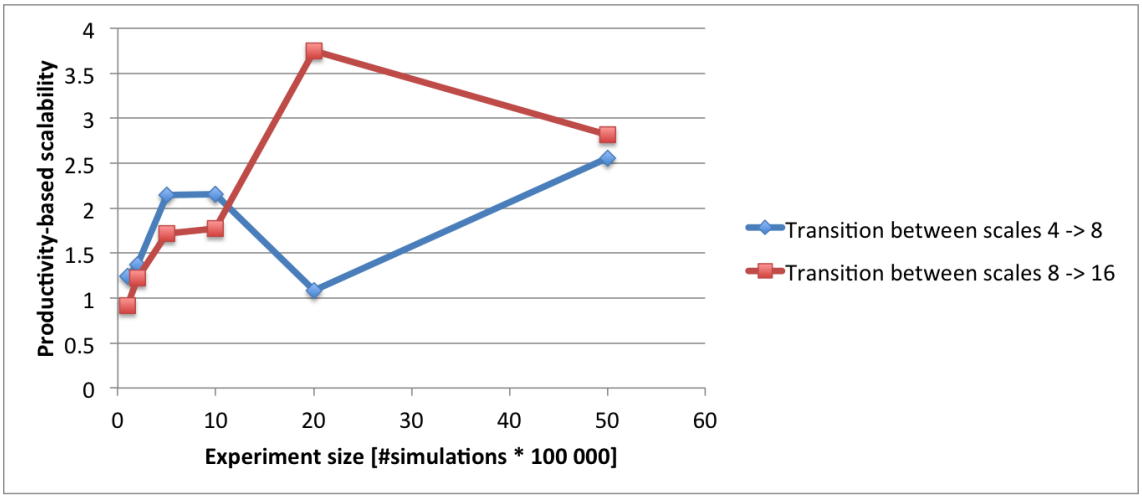

Figure 6.6: Productivity-based scalability for different experiment sizes.

Comparison of efficiency- and productivity-based scalability metrics When comparing both types of scalability, i.e. efficiency-based and productivity-based scalability, many similarities can be noted. Both curves behave in a similar fashion, i.e. values for smaller experiments are similar for both transitions and scalability increases along with experiment size. However while efficiency-based scalability remains between 0.7 and 1.3, with an average value of 0.91, the values of productivity-based scalability are between 0.9 and 3.7, with an average value of 1.89. In the context of efficiency this means additional resources are effectively used, albeit with an upper limit (since the average value is less than 1). In the context of productivity this means that larger scales ensure higher productivity. This is related to higher response values and lower cost of computation when conducting experiments with more than 1 000 000 simulations. Moreover, as the average value is greater than 1, there does not appear to be an upper productivity limit.

Using real-life simulations in scalability evaluation One can argue that using the foo simulation, which includes no calculations, is inappropriate for scalability testing. Therefore the author estimated the number of Simulation Managers necessary to saturate Scalarm by running actual simulations. This estimation was performed by applying the platform throughput metric, which provides information on how many simulations can be scheduled per second, as well as the time necessary to calculate sample real-life simulations.

In the course of evaluating Scalarm, several real-life simulations were executed. Two of them are described below:

- In the context of the EDA EUSAS project, which is described in Chapter 7,

a multi-agent simulation with the goal of supporting the training process of security forces was developed. A sample simulation scenario involved controlling access of civilians to a military base camp during elections in a foreign deployment scenario. The simulation had 92 input parameters describing the initial emotional state and other attributes of simulated entities. The average execution time of a simulation was approximately 3 minutes.

- The second simulation used to evaluate the Statistically Similar Representative Volume Element (SSRVE) of a given microstructure, developed in the context of the PLGrid Plus project. SSRVE is typically used for decreasing the cost of multiscale microstructure modeling by reducing the number of finite elements required to generate a reliable mesh. This application had about 20 input parameters describing features of the microstructure and properties for optimization algorithms. The average execution time of this simulation was approximately 11 minutes.

An average throughput for configuration (1, 1) was about 4776 simulations scheduled per minute, based on data from Table 6.4. By using this throughput value and information about execution of real-life simulations, we estimated the number of parallel Simulation Managers necessary to generate such a workload (Table 6.7).

Table 6.7: Estimated number of Simulation Managers necessary to saturate the Scalarm platform using configuration(1, 1) and real-life simulations.

Estimated Simulation Manager count for scalability tests with real-life simulations		
Real-life simulation	Measured simulation execution time [min]	Estimated Simulation Manager count
Multi-agent security forces simulation	3	14 328
SSRVE simulation	11	52 536

Even assuming that each Simulation Manager needs only one CPU core, the required resources are still extensive. This was the main reason for performing scalability evaluation without actual computations.

6.3 Self-Scalability Evaluation

The second part of the evaluation process of the Scalarm platform concerns the self-scalability feature. Unlike performance metrics, which describe scalability, this process aims to highlight the features which differentiate Scalarm from other existing solutions, i.e. the capability of the platform to scale itself based on scaling rules. Instead of trying to implement generic scaling rules which would fit all needs, the author implemented a generic mechanism of executing different scaling actions

based on externally defined scaling rules. This is a conscious design choice, undertaken to avoid constraining the scalability behavior of the platform while enabling administrators to control and adjust said behavior to actual simulations and deployment conditions. Scaling rules are an essential part of this mechanism since they enable Scalarm administrators to express requirements regarding scaling in a machine-processable form. Each defined scaling rule is then handled by a dedicated component within a self-scalable service, called the Scalability Manager which monitors the service and executes scaling actions when necessary.

6.3.1 Testing scenario

Regarding the self-scalability feature, the main purpose of the testing scenario is to study how Scalarm behaves when faced with a varying number of clients in a given period of time. Hence, instead of adjusting resource configurations manually, we set up a number of machines and started Scalarm on only two of them. In addition, we defined scaling rules to control Scalarm behavior. We then proceeded to add and remove clients dynamically in a strictly defined, repeatable order.

As a result, we could evaluate how Scalarm responds to varying workload on its own, relying on the predefined scaling rules. In addition, this scenario enabled us to compare the cost differences between running a self-scaling Scalarm installation and a fixed-configuration deployment.

The number of clients (Simulation Managers) changes over time as follows:

1. t_0 - the test is started
2. $t_1 = t_0 + 10\,minutes$ - 120 clients are started - the total number of clients is 120
3. $t_2 = t_1 + 5\,minutes$ - 120 clients are started - the total number of clients is 240
4. $t_3 = t_2 + 10\,minutes$ - 240 clients are started - the total number of clients is 480
5. $t_4 = t_3 + 10\,minutes$ - 480 clients are started - the total number of clients is 960
6. $t_5 = t_4 + 10\,minutes$ - 480 clients are stopped - the total number of clients is 480
7. $t_6 = t_5 + 10\,minutes$ - 240 clients are stopped - the total number of clients is 240
8. $t_7 = t_6 + 10\,minutes$ - 240 clients are stopped - the total number of clients is 0

9. $t_8 = t_7 + 10\,minutes$ - the test concludes

A single test run took 75 minutes to complete and number of clients (i.e. Simulation Managers) varied between 0 and 960. It should be noted that Simulation Managers merely schedule simulations and do not perform any computations on their own. Hence, only the management part of Scalarm, i.e. Experiment and Storage Managers, was loaded for this test.

We ran this test with three configurations:

- with no scaling rules defined - Section 6.3.2,
- with scaling rules defined for Experiment Managers only - Section 6.3.3,
- with scaling rules defined for both Experiment and Storage Managers - Section 6.3.4.

During tests, Scalarm had access to 8 machines (described earlier on in this chapter) upon which to run component instances.

6.3.2 Self-scalability test - scaling rules disabled

We first launched the described testing scenario using 4 machines to run Experiment Managers and 4 machines to run Storage Managers. The goal was to measure the highest achievable performance (in terms of simulations scheduled) with the highest corresponding platform cost. In this configuration during a single test run Scalarm managed to schedule 499 292 simulations on average (Table 6.8). During each test two random physical servers were monitored, i.e. one with an Experiment Manager instance and one with a Storage Manager instance. Since the workload was spread evenly across all machines, there were no significant differences between the monitored metrics on different machines. Fig. 6.7 presents CPU load [%] on an Experiment Manager machine while Fig. 6.8 depicts the 'wait for I/O request to complete' metric [ms] on a Storage Manager machine. This storage-related metric summarizes the time spent in the I/O queue and the time of executing the actual request.

Regarding CPU load, utilization was close to 0% in the first 10 minutes and the last 10 minutes since no clients were running during that time. Throughout the experiment CPU utilization peaks when new clients are started, but in most cases remains below 60%. This indicates that the selected resource configuration is underloaded. On the other hand, HDD utilization on the Storage Manager machine exhibits shorter but more frequent peaks. This is the result of the database management system periodically committing its journal to the local disk. Otherwise the local disk is relatively underutilized since all operations are done in main memory.

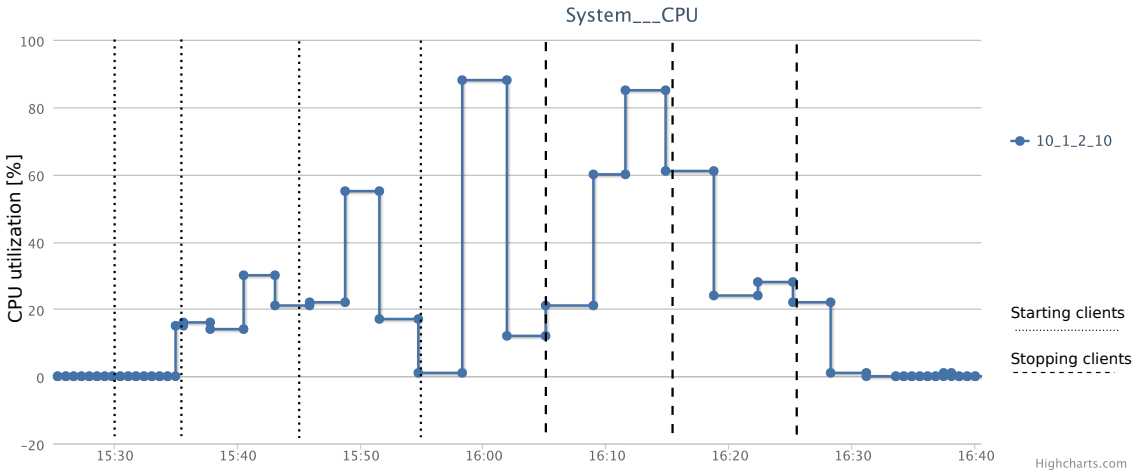

Figure 6.7: CPU load [%] on an Experiment Manager machine - test with no scaling rules.

To determine the cost of our configuration we calculated the number of CPU-minutes used across 8 servers, i.e. 8 * 75 = 300 minutes, and multiplied this by the cost of a CPU-minute based on Amazon EC2 pricing, i.e. $1.160 per CPU-hour. Consequently, the full resource configuration test (with no scaling rules) works out to approximately $5.80; however monitoring metrics indicate that machines were underloaded most of the time.

6.3.3 Self-scalability test with scaling rules for the Experiment Manager

An ideal situation would involve balancing the workload evenly across all available servers, e.g. to achieve 80-90% utilization, but only when there is actual workload. Thus, we defined scaling rules to adjust the Scalarm resource configuration dynamically, i.e. Scalarm should shut down unnecessary instances and release computational resources, lowering the cost of the platform. The first attempt to achieve this goal is by defining scaling rules for the Experiment Manager only. Since both up- and downscaling need to be taken into account, we specified the following rules in the context of the monitored machine:

1. if the average CPU load exceeds 80% over a period of 90 s then start a new Experiment Manager instance
2. if the average CPU load is lower than 20% over a period of 240 s then stop a running Experiment Manager instance

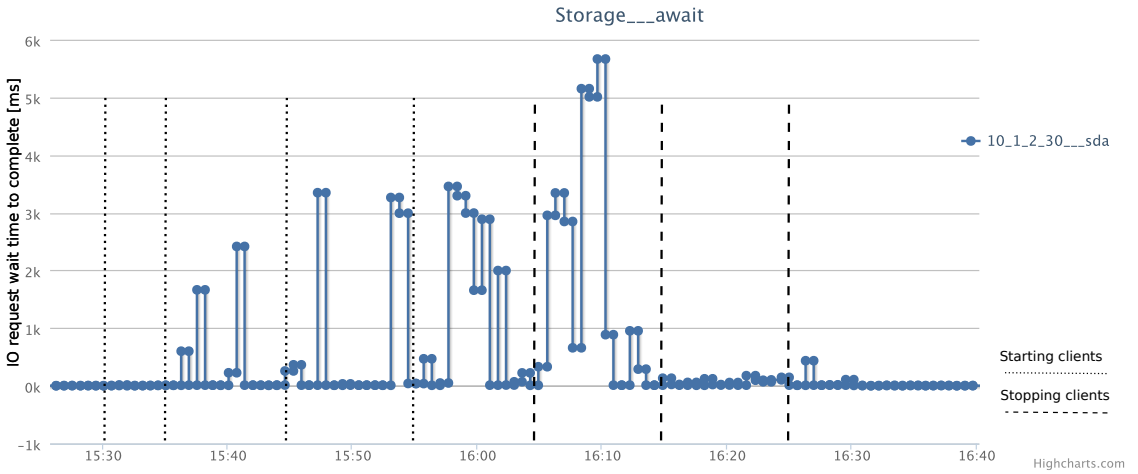

Figure 6.8: Wait time for I/O request to complete [ms] on a Storage Manager machine - test with no scaling rules.

It is worth noting that every scaling action, e.g. starting or stopping a component instance, is followed by a "cooldown" period, i.e. an interval during which scaling rules are disabled. By introducing such a period we intend to eliminate any negative influence of scaling actions upon the platform's workload. In the following test the "cooldown" period was set to 5 minutes. For the purpose of this evaluation we allowed only one type of Scalarm component per physical server at any given moment. This condition facilitates workload analysis. At the beginning of the test we ran only two machines, one with an Experiment Manager instance and one with a Storage Manager instance. This is the minimal resource configuration which fulfills the conditions presented above. Other servers are utilized only as a result of scaling actions triggered by scaling rules.

Given this configuration and using the defined scaling rules, Scalarm managed to schedule (on average) 375 951 simulations throughout the test (Table 6.8). Measurements of system metrics are depicted in Fig. 6.9 (CPU load) and Fig. 6.10 (storage load) respectively.

Regarding CPU load, new Experiment Manager instances come online two minutes or so following peak load conditions. As a result, CPU load drops significantly in each case. However, if, for some reason, CPU load remains low for a few minutes, an Experiment Manager instance is shut down. At the end of the test, when there are no clients at all, Scalarm reverts to its minimum required configuration, i.e. two servers.

On a server with a Storage Manager instance the utilization level of the local disk seems to be uncorrelated with CPU load. Instead, it is related to the number of scheduled simulations, i.e. periodic increases in disk load can be noted as a result

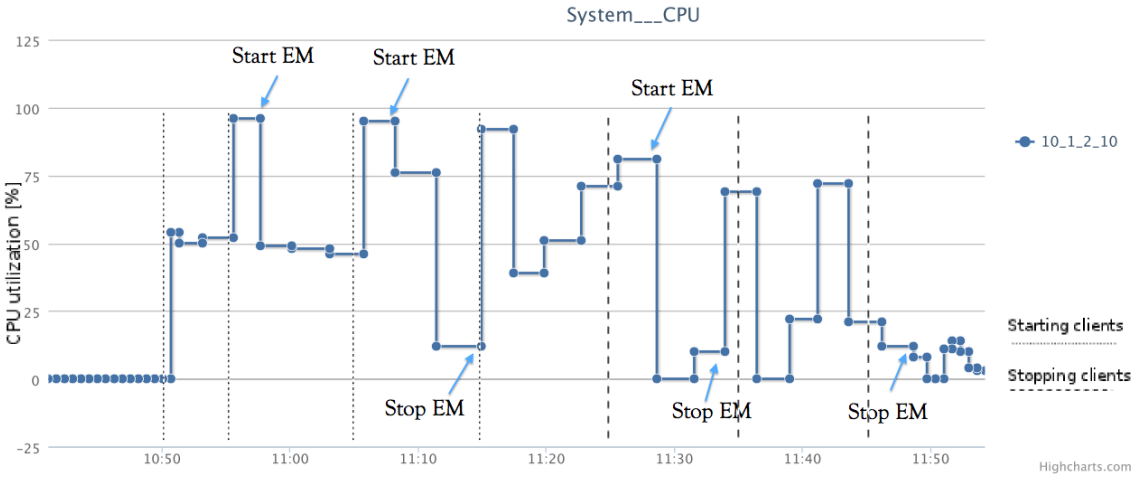

Figure 6.9: CPU load [%] on an Experiment Manager machine - test with scaling rules for the Experiment Manager.

of database journal writes. Between these peak load conditions, disk utilization is almost negligible.

To calculate the average cost of running this test we aggregated the number of CPU-minutes used by our configuration, which in this case was 213. By referring to the cost of a CPU-hour for a Cloud virtual machine, we can estimate the final cost as $4.12.

6.3.4 Self-scalability test with scaling rules for Experiment Managers and Storage Managers

Previously, Storage Managers were run using a static configuration, which led to either underutilization of resources (with 4 machines in the configuration), or to decreased performance (with only 1 machine in the configuration). The following test takes self-scalability management a step further by specifying separate rules for Experiment and Storage Managers, as follows:

1. if the average CPU load exceeds 80% over a period of 90 s then start a new Experiment Manager instance
2. if the average CPU load is lower than 20% over a period of 240 s then stop a running Experiment Manager instance
3. if the average wait time for storage exceeds 2000 ms over a period of 240 s then start a new Storage Manager instance

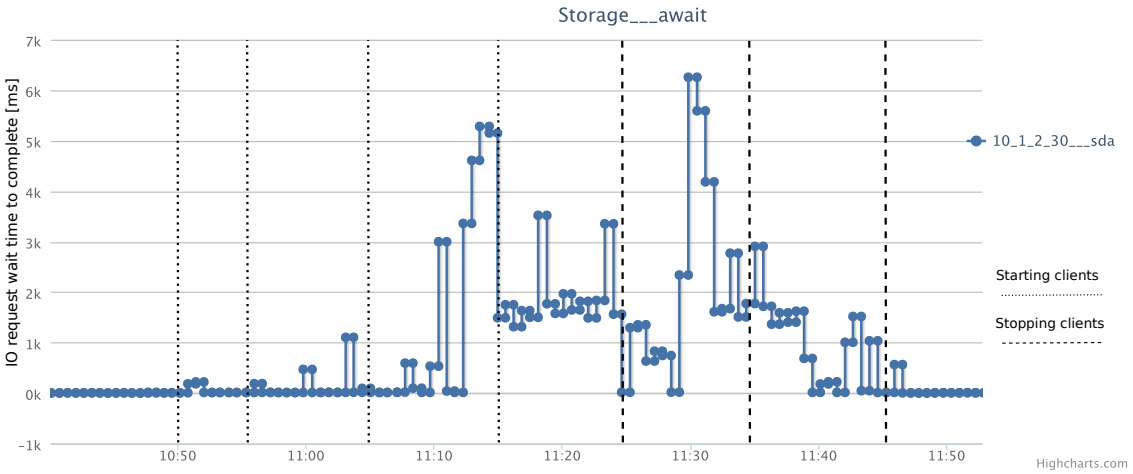

Figure 6.10: Wait time for I/O request to complete [ms] on a Storage Manager machine - test with scaling rules for the Experiment Manager.

4. if the average wait time for storage is lower than 200 ms over a period of 300 s then stop a running Storage Manager instance

Similarly to the previous test, we initially ran only two machines and added other machines on demand, as a result of scaling actions. The "cooldown" period was again set to 5 minutes. In this configuration and with the provided scaling rules Scalarm managed to schedule 454 059 simulations during the whole test (Table 6.8). System metric measurements are depicted in Fig. 6.9 (CPU load) and Fig. 6.10 (storage load).

Similarly to the previous test, Experiment Managers were started approximately two minutes following peak load conditions. In the middle of the test an Experiment Manager instance was stopped, probably due to performance issues involving the Storage Manager. However, in contrast to the previous test, a new Storage Manager instance was started in the second half of the test. Interestingly, this operation increased the disk load on the Storage Manager machine until the end of the test. This is related to the behavior of MongoDB, which is utilized as the Storage Manager backend. When a new instance of MongoDB is added to a cluster, some data from other instances is transferred to the new instance, which generates load on the local disk. This additional workload persists until the end of the test, hence the additional Storage Manager also survives until the end, instead of being shut down when there are no more clients. Surprisingly, no more Storage Managers are started even if the storage workload exceeds the threshold defined in scaling rules. This is a side effect of the "cooldown" period, i.e. other scaling actions, namely stopping Experiment Managers, prevent additional Storage Manager instances from being started.

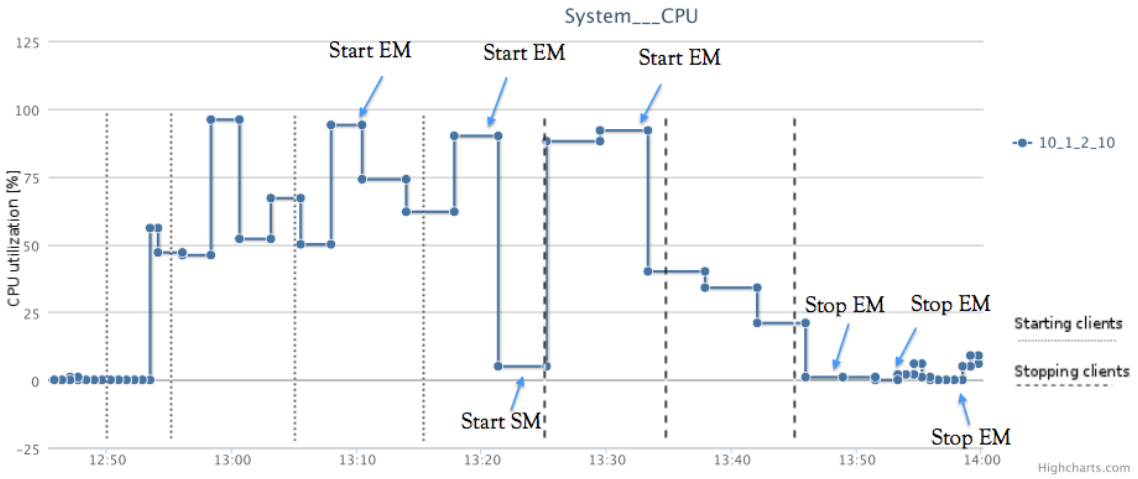

Figure 6.11: CPU load [%] on an Experiment Manager machine - test with scaling rules for all components.

Under these scaling rules, the Scalarm configuration used 246 CPU-minutes, which means that the average cost of the experiment was $4.76.

6.3.5 Self-scalability evaluation conclusions

The conducted tests confirmed the functionality of Scalarm regarding support for scaling rules. Administrators can easily manage resources dedicated to Scalarm, simply by specifying how the platform should scale. This mechanism enables Scalarm to be started with very few resources at first and then expand the resource configuration on demand. An additional reason to use the self-scalability feature is cost-effectiveness. In our case cost-effectiveness can be expressed by the number of simulations scheduled per $1. Information regarding cost-effectiveness, obtained from the presented tests, is collated in Table 6.8.

Table 6.8: Cost-effectiveness associated with the self-scalability feature.

Cost-effectiveness of self-scalable Scalarm			
Test description	Scheduled simulations	Total cost [$]	Scheduled simulations per $1
No scaling rules	499 292	5.80	86 084
Scaling rules for Experiment Manager	375 951	4.12	91 250
Scaling rules for Experiment and Storage Manager	454 059	4.76	95 390

The least cost-effective configuration was the one without any scaling rules, while

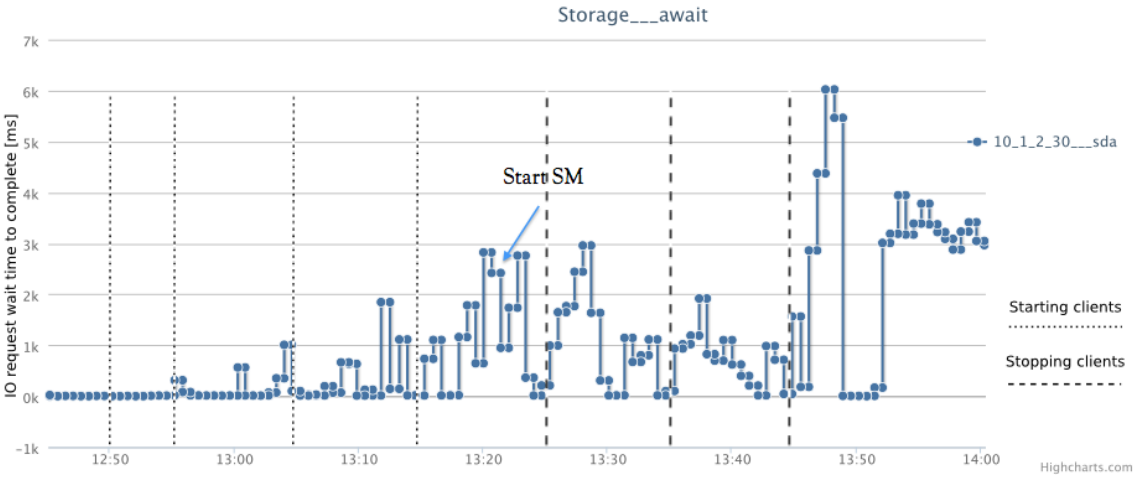

Figure 6.12: Wait time for I/O request to complete [ms] on a Storage Manager machine - test with scaling rules for all components.

the other two configurations enabled scheduling more simulations per $1. Although this factor heavily depends on the actual load characteristic, the author intended to model a representative use case, with quiescent periods as well as peak load periods. Many research studies suggest that modern data centers are relatively underloaded. In such cases, running a static Scalarm configuration would be even less cost-effective than utilizing self-scalability. Thus, self-scalable software can provide significant savings terms of hardware maintenance while at the same time increasing the resource utilization level.

Chapter 7

Data Farming Utilization in Training of Security Forces

This chapter discusses the application of Scalarm in the EDA EUSAS project. In particular, the role of data farming is presented and the motivation for using Scalarm is provided. Scalarm features essential for the project are described in the context of a sample data farming experiment. Scalarm support for different experiment process phases (DoE, simulation execution and output analysis) is also discussed.

7.1 Problem Description and Motivation for Data Farming Usage

The features of the Scalarm platform were first evaluated in the European Defence Agency (EDA) EUSAS project [14], which stands for European Urban Simulation for Asymmetric Scenarios. The main project objective was to enhance training of security forces, such as the military and police, in asymmetric scenarios where the forces in opposition are not conventional, i.e. one force has a structured hierarchy and rules of engagement while the other is an unstructured group which uses unconventional means and tactics. Asymmetric threats in urban settings involve the operation of a relatively small group of soldiers in a city with a civilian population ranging from neutral to hostile. Furthermore, civilians usually outnumber security forces and therefore special training is needed to effectively handle this kind of situation.

The project involved researchers from computing institutes in Poland, France, Germany, Slovakia, Slovenia and Sweden. It was coordinated by a defense and security company called CASSIDIAN. The project was motivated by a number of disruptive events in large cities, e.g. riots in Serbia following the proclamation of independence by Kosovo in 2008, or Iranian election protests in 2009 which caused numerous casualties both among the security forces and civilians.

During the project multiple behavioral modeling techniques were used [115] to

analyze, evaluate and enhance security force tactics. In addition, the process was supported by a large-scale computational infrastructure provided by ACC Cyfronet AGH, especially the "Zeus" cluster, combined with resources from public Clouds, e.g. Amazon EC2.

The rationale behind utilization of the data farming methodology in the project was related to the complexity of the analyzed phenomena. Use case scenarios involve numerous entities, e.g. civilians and security forces. Each entity can be described by a number of parameters describing their emotional state, behavioral characteristics or physical features. Hence, simulation scenarios involve large parameter spaces and nontrivial dependencies between the simulated entities which are nearly impossible to model using an analytical approach. In addition, data farming has been known to successfully handle similar cases involving military simulations.

7.2 Solution Overview

The EDA EUSAS project intended to enhance security force training with human behavioral analysis techniques. In addition, statistical analysis was used to simulate various scenarios to improve force strategies. An important part of the EDA EUSAS project was Human Behaviour Modelling (HBM) with Agent Based Simulations (ABS) [116]. Each civilian and soldier taking part in a mission was modeled as an independent agent with multiple parameters, e.g. emotional state and the ability to interact with other agents.

An overview of the process is depicted in Fig. 7.1. Key steps of the process are as follows:

1. "Mission Guidelines Creation & Adjustment" - the first step of the training enhancement proposed by the project. In this step, basic guidelines are drafted for specific missions. During subsequent iterations of the process these guidelines are adjusted to acknowledge new information regarding agent strategies.

2. "Adapt Agent Models (Soldier & Civilian)" - adapting agent models, both for security forces and civilians, to specific aspects of the analyzed mission using manual input (e.g. expert guidelines) as well as automatic calibration.

3. "Adapt Simulation Scenarios" - changing simulation scenarios to better reflect the mission environment, e.g. terrain, buildings, or the number of security agents and civilians.

4. "Calibrate Models" - tuning agent models by executing simulations multiple times manually. Based on the gathered data, agent models and simulation scenarios can be fine-tuned.

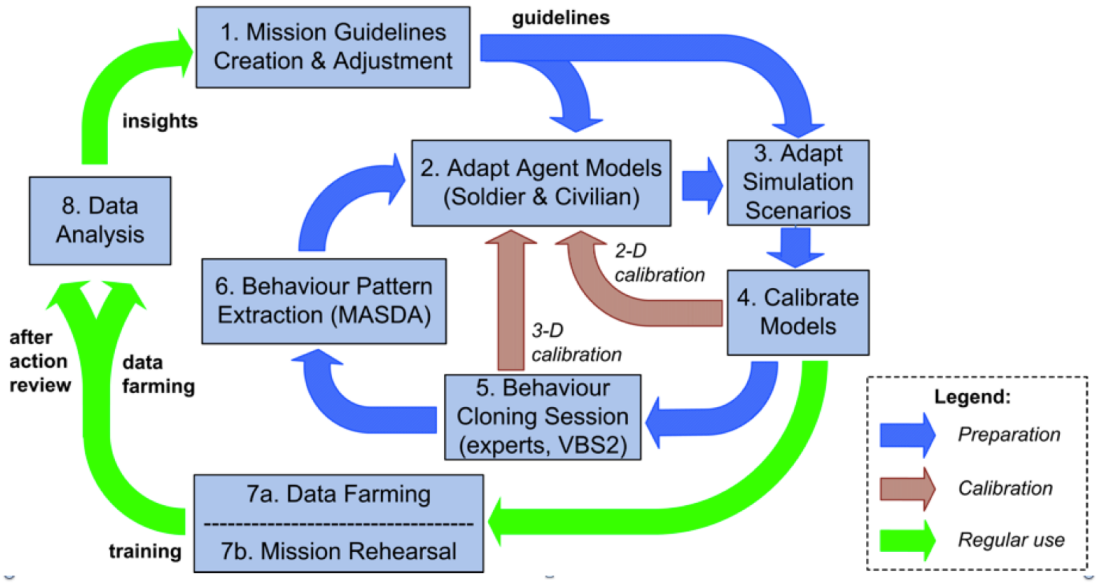

Figure 7.1: Improving security force training in the EDA EUSAS project [14].

5. "Behaviour Cloning Session" - a special training session with real soldiers in virtual reality. Each soldier participating in the session supervises his/her virtual avatar in the simulated mission, which is repeated multiple times to explore the possible options. Each mission run is recorded for further analysis of selected strategies and behaviors.

6. "Behaviour Pattern Extraction (MASDA)" - analysis of recorded missions and semiautomatic extraction of patterns reflecting soldiers' behavior. The project developed an application for this task called MASDA [117], which is a pattern recognition tool dedicated to behavioral analysis. Extracted patterns are used to replace real soldiers with fully automated virtual agents.

7. a) "Data Farming" - conducting data farming experiments using the previously calibrated agent models instead of real security forces. The main objective is to generate large amounts of data describing soldiers' behavior by using computer simulations. Due to the large parameter space, it is impossible to evaluate all possible mission options using real soldiers. Hence, cloned behavior is used in this process.

8. b) "Mission Rehearsal" - simulation of several interesting mission cases using the calibrated models. The main objective is to perform in-depth analysis of selected cases to identify any undesirable elements in the simulation. It is a

complementary approach to data farming in which numerous simulations are executed to analyze a large spectrum of cases statistically.

9. "Data Analysis" - utilizes all data gathered in the data farming and Mission Rehearsal steps to extract knowledge about possible vulnerabilities in security force strategies. Based on the extracted knowledge, recommendations regarding training improvements are proposed.

The above process is iteratively repeated until no further behavioral improvements are necessary. Each iteration involves one or more data farming experiments to analyze soldiers' behavior in a vast set of possible mission configurations. Simulation scenarios used in data farming experiments concern real security force missions, e.g. crowd control, which cannot be easily trained in real-world conditions.

Simulation models treat civilians and soldiers as agents. During successive iterations these models are calibrated to match particular mission aspects. Each simulated agent is described by many parameters, such as emotional state, starting location or intentions. The parameters of all simulated agents, along with parameters describing the simulated environment, constitute input for the data farming experiments. Experiment output includes different MoEs specific for the given mission, e.g. the number of injured soldiers, average level of civilian aggression, etc.

Due to the large parameter space, the following DoE methods were implemented and used during the project [118]:

- *Near Orthogonal Latin Hypercubes* (NOHL) [119] is an experiment design method which can be represented by a matrix with n rows (denoting separated simulation input vectors) and k columns (denoting input parameters with uniformly distributed possible values). In addition, each pair of distinct columns has zero correlation, i.e. their inner product is zero.

- *Full factorial* is a design involving k input parameters each with different possible values. Its output constitutes all possible combinations of input parameter values.

- *Fractional factorial* is similar to full factorial but provides only a subset of possible combinations. A special case of the fractional factorial is the 2^k method, which only considers the maximum and minimum values for each input parameter.

Support for heterogeneous computational infrastructures was another requirement, dictated by the complexity of simulated scenarios and their large parameter space. Even when using very restrictive DoE methods, e.g. 2^k, numerous simulations has to be executed, potentially exceeding the capacity of a single data center. Therefore the computational infrastructure for data farming needed to include resources

from different providers, e.g. institutional clusters, national Grid environments and commercial Clouds. For the purpose of the project we selected the PL-Grid infrastructure as a national Grid environment and Amazon EC2 as a sample commercial Cloud provider.

To support data exploration the following visualization methods were implemented [120]:

- Scatter plots, which describe pairs of selected MoEs and/or input parameters. A scatter plot simply displays the values of two variables as points in a 2D Cartesian space. It can be useful for finding relationships or correlations between any two variables.

- A histogram chart, which, for a selected MoE, shows how many simulations ended with each of the possible values. The user should be able to specify how many regions the MoE value should be divided into, thereby controlling the resolution of the chart. In addition, this type of analysis should include some basic statistical information about the selected MoE, i.e. minimum, mean, maximum and standard deviation values.

- Regression trees – a commonly used method of discovering input parameters with particularly high influence on the output. A regression tree is created for a specific MoE and measures the influence of input parameters upon the selected MoE, i.e. the change of MoE values for a given input parameter (compared to other input parameters). For the selected MoE, a regression tree is a binary tree where each node (except leaves) contains an inequality. The left-hand side represents the name of an input parameter while the right-hand side comprises a given value and the number of experiment instances covered by the node. Each node (including leaves) lists the mean value for the selected MoE calculated on the basis of completed simulations – specifically, these simulations which satisfy all inequalities occurring along the path from the root node to the given tree node. At the root node, the number of covered experiment instances equals the number of experiment instances completed since the beginning of analysis.

7.3 Functionality Evaluation

The author conducted an experiment to demonstrate essential Scalarm features regarding data farming. The simulation involved controlling the access of civilians to a military base camp during elections in a foreign deployment scenario. Two groups of civilians have gathered at an entrance to a base of operations with the intention to start a skirmish. From the security forces' point of view the goal of

this scenario is to prevent escalation of aggression by way of effective negotiations. Civilians may respond in various ways depending on their input parameter values. Consequently, actions performed by security forces should be adjusted as necessary. The input parameters for this simulation scenario describe:

- the emotional state of civilians (e.g. agitation) at the start of the scenario,
- behavioral characteristics, e.g. propensity for aggression and fear of security forces,
- ability to affect others, e.g. prestige and influence radius.

The goal of this data farming experiment was to minimize the number of injuries among civilians and soldiers regardless of the civilians' behavior. The author decided to parametrize 14 input parameters (out of 92), describing:

- the size of civilian groups,
- the agitation level at the the beginning of the scenario,
- civilians' propensity for aggression,
- the prestige of two civilian leaders (one leader per group).

The output of each simulation included a text file with less than 7 MB of simulation logs (on average) and 44 different MoEs describing the aggregated emotional states of different entity groups and statistic regarding the simulated scenario – in particular, the number of injured agents.

At the experiment design phase the above listed parameters were set to use "range" parametrization using the fractional factorial method described above, while all other parameters retained their default values. As a result, 16 386 different combinations of input parameters were generated.

One of the key Scalarm features regarding data farming is monitoring the progress of simulations. A system screenshot of this view is depicted in Fig. 7.2. In particular, the user can monitor various statistics of the experiment, e.g. the pace of simulation execution, the number of simulations already executed and the number of pending simulations.

To evaluate support for heterogeneous computational infrastructures, the following resources were used to compute actual simulations:

- 9 worker nodes from a private cluster,
- 50 Grid jobs scheduled upon the PL-Grid infrastructure, particularly the "Zeus" cluster at ACC Cyfronet AGH,

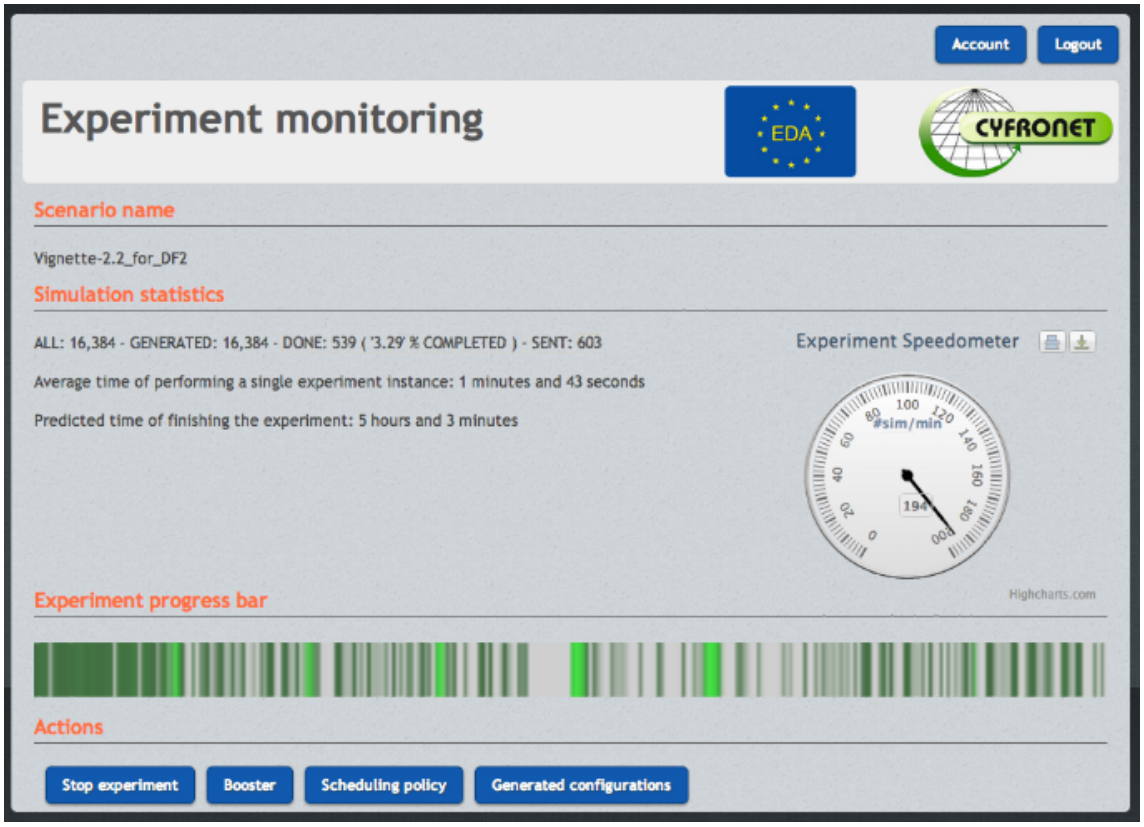

Figure 7.2: The progress monitoring view of a data farming experiment in Scalarm.

- 50 High-CPU Extra Large instances from Amazon EC2.

By using this set of resources more than 620 simulations were executed simultaneously, with more then 140 simulations completing in each minute. The average execution time of a single simulation was about 2 minutes.

Another useful feature of Scalarm is built-in support for various statistical analyses which can rely on completed simulation results: histograms, regression trees and bivariate graphs. Basing on ongoing statistical analysis of precomputed simulations the user can specify additional values of input parameters to compute. Hence the user can conduct the experiment in an exploratory way, i.e. start with a small parameter space and then enlarge it as needed.

As part of the demonstration the author performed sample analysis using regression trees, having previously completed nearly 800 simulations, as depicted in Fig. 7.3. Regression trees are often used to identify dependencies between selected MoE and input parameters. In our case the most significant input parameter (affecting the number of civilians and soldiers killed) was the size of the civilian group responsible for the skirmish. As this input parameter had only two possible values,

Figure 7.3: Regression tree analysis view for partial experiment results in Scalarm.

i.e. 1 and 50 (as specified by the fractional factorial method), the analysis result was not a surprise.

However, the user can investigate this parameter in more detail by using the built-in ability to add more input parameter values on the fly. This Scalarm feature is depicted in Fig. 7.4. The user can review existing values and specify additional value ranges.

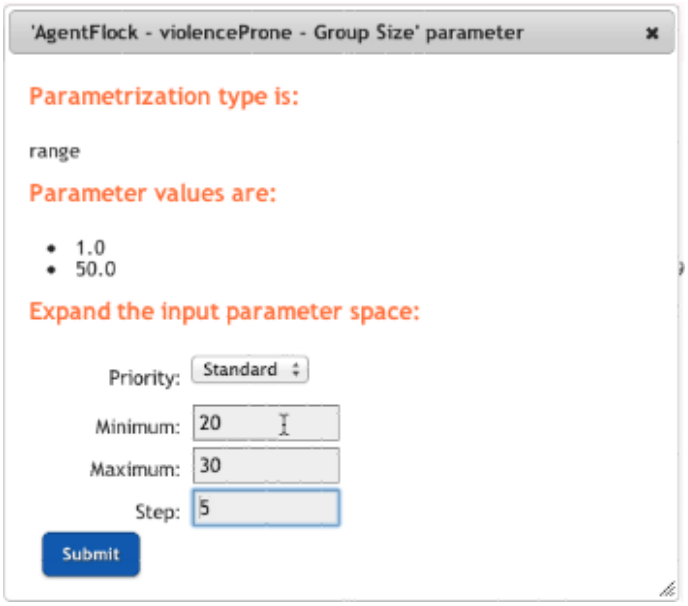

Figure 7.4: Experiment parameter space extension dialog in Scalarm.

Chapter 8

Conclusions and Future Work

In this chapter the author summarizes the thesis by presenting the achieved goals and contributions. Areas which can benefit from the thesis are discussed. Finally, interesting directions for future work are listed.

8.1 Summary

In this work the author explored the application of self-scalability to large-scale software platforms in the context of the data farming methodology. Support for heterogeneous computational infrastructures was an important aspect of this work. As modern computational infrastructures (such as Clouds) tend to offer scalability in the infrastructure layer as well as elastic cost management, the problem of building software capable of exploiting these features is highly relevant.

The author proposed an extension of the SOA paradigm called self-scalable services, along with a formal way of expressing knowledge on how a given service should scale itself when necessary. Based on these concepts a massively self-scalable platform for data farming, called Scalarm, was presented. Scalarm includes a reference implementation of both concepts to support large-scale data farming experiments with heterogenous computational infrastructures.

The stated thesis of this work was proved by way of experimental evaluation of non-functional and functional aspects of the platform. In particular, synthetic tests of massive scalability and self-scalability of the platform were performed. Self-scalable services provide scalable throughput which surpasses the capabilities of existing solutions. By using automatically managed scaling rules the platform is more cost-effective than manually managed deployments. Functional aspects of the platform were evaluated in the context of the EDA EUSAS project which utilized Scalarm to conduct data farming experiments which support training of security forces.

8.2 Research Contribution

The contribution of this thesis can be summarized as follows:

- In order to manage software scalability, the concept of scaling rules was introduced as a way to express knowledge related to the scaling behavior of software, i.e. conditions and actions concerning scalability. Although scaling rules are specified by users, they can be processed automatically by dedicated software. As a result the platform is enriched with the self-scalability feature.
- An extension of Service-Oriented Software, called self-scalable services, was proposed in order to extend software (composed from loosely-coupled services) with transparent, built-in scaling mechanisms. This extension facilitates development of massively scalable platforms while leveraging the benefits of the service-oriented approach.
- To evaluate both concepts, a dedicated platform for conducting data farming experiments, called Scalarm, was implemented and tested. Results of this experimental evaluation, described in Chapters 6 and 7, proved both aspects of the research hypothesis stated in Section 1.6, i.e. extensive scalability achieved using self-scalable services, and cost-effectiveness achieved by utilizing self-scalability.

8.3 Potential Areas of Application

The concepts described above were utilized to develop a platform for data farming experiments in the area of security force training. Note that all presented concepts and design features are generic enough to be applicable to many other areas. In fact, each contribution of this thesis may be applied separately to solve unrelated problems, as described below.

- Scaling rules are a suitable mechanism for any system which has to automatically adjust to changes in its environment. This description corresponds to almost all Cloud applications whose workload depends on the number of clients. The relation with Cloud environments determines the actual cost of operating a system, which depends on the number of utilized CPU-hours. Our experimental evaluation proved that support for scaling rules can significantly decrease the cost of running software in such environments.
- Self-scalable services are dedicated for platforms composed from multiple independent services, each of which needs to be scaled in a transparent manner, starting from a single laptop up to dozens of servers.

- Scalarm is a generic system which can run any data farming experiment and, in fact, any parameter study experiment, i.e. multiple simulations with slightly different input parameter values. Besides preparing the input parameter space using various DoE methods, Scalarm facilitates interaction with heterogeneous computational infrastructures and therefore minimizes the effort needed to redeploy standalone applications to Grid or Cloud environments. In particular, Scalarm is well suited for task farming [13], i.e. executing large numbers of tasks using heterogeneous computational resources.

8.4 Future work

Although Scalarm is a fully functional platform, much remains to be done. A list of interesting aspects which merit further investigation is presented below.

- Currently, scaling rules have to be defined manually. This task usually falls to administrators, since they are the ones with detailed knowledge about the available infrastructure and workload generated by users. When the workload pattern changes, existing scaling rules may become obsolete and it would be desirable to develop an autonomous platform which can discover scaling rules automatically based on the observed user behavior. This would allow the platform to adapt to changing user needs without human intervention.

- Another important aspect is platform genericity, i.e. the ability to run any type of simulation regardless of the programming language, resource requirements or input data formats. This is a hard problem, but one which needs to be tackled if the platform is to be widely adopted.

- In this thesis scaling rules were presented in the context of Experiment and Storage Managers; however Scalarm includes another important component, i.e. the Simulation Manager, which encapsulates simulation execution. The number of Simulation Manager instances is currently predetermined by the analyst who performs the actual experiment. The purpose of this decision is to enable users to choose which type of infrastructure Simulation Managers will actually run on. This is important since supporting e.g. Amazon EC2 can incur real costs for end users. However, in general, the platform should adjust the number of Simulation Manager instances automatically, relying on user-defined rules. In addition, the platform should be able to adjust the ratio of resources dedicated to the "master" part and the "worker" part while the pool of available resources remains fixed.

- An important aspect of executing simulations in Cloud environments is cost. Commercial Clouds often comprise different types of resources with varying

cost per hour. For example, a virtual machine with many CPU cores costs more but also provides more computational power than a single-core unit. Cost minimization in such environments is widely discussed in recent literature [96]. Scalarm could present the user with suggestions regarding optimal Cloud resource allocation for a given simulation. Furthermore, the cost of data storage and data transfer should be taken into account when providing such a proposal.

- Last but not least, the data storage aspect requires further analysis of data and computation locality issues. Data farming experiments often depend heavily on efficient access to data. When using heterogeneous computational infrastructures, the problem of data transfer to and from computational resources becomes highly relevant. Resources from one computational infrastructure can be inaccessible to another infrastructure. Moreover, transferring data between geographically distributed locations can be inefficient. Techniques which involve shifting computation closer to data (instead of the other way around) are commonly referred to as *computational storage*. Awareness of were the data is actually stored facilitates optimal task scheduling. Scalarm might utilize this knowledge when deciding how to distribute simulations between different computational systems. Furthermore, data management techniques proposed by the author in [121, 122, 123] could by exploited with regard to Cloud environments.

Abbreviations and Acronyms

Abbreviation	Explanation
ABS	Agent Based Simulations
AC	Autonomic Computing
ACC	Academic Computer Centre
API	Application Programming Interface
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
CSV	Comma-Seperated Values
DIRAC	Distributed Infrastrcuture with Remote Agent Control
DNS	Domain Name Service
DoE	Design of Experiment
EC2	Elastic Compute Cloud
EDA	European Defence Agency
EU	European Commision
EUSAS	European Urban Simulation for Asymmetric Scenario
FLOPS	FLoating-point Operations Per Second
GbE	Gigabit Ethernet
GUI	Graphical User Interface
HBM	Human Behaviour Modelling
HCI	Human Computer Interface
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
HTTP	HyperText Transfer Protocol
HTC	High Throughput Computing
IaaS	Infrastructure as a Service
IMDG	In-Memory Data Grid
IPC	Inter-Process Communication
IST	User-friendly Information Society

JWARS	Joint Warfare System
LHC	Large Hydron Collider
LHCb	Large Hadro Collider beauty
MoE	Measure of Effectiveness
MQ	Message Queuing
NASA	National Aeronautics and Space Administration
NFS	Network File System
NOHL	Near Orthogonal Latin Hypercube
OLTP	On-Line Transaction Processing
OMD	OldMcData
PaaS	Platform as a Service
PBS	Portable Batch System
POIG	Innovative Economy Program
QoS	Quality of Service
RAM	Random Access Memory
RDBMS	Relational DataBase Management Systems
REST	Representational State Transfer
S3	Simple Storage Service
SaaS	Software as a Service
SAN	Storage Area Network
SDK	Software Development Kit
SEDA	Staged Event-Driven Architecture
SEED	Simulation Experiments & Efficient Designs
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SPOF	Single Point Of Failure
SSH	Secure SHell
SSRVE	Statistically Similar Representative Volume Element
SQL	Structured Query Language
XAP	eXtreme Application Platform
XML	eXtensible Markup Language

Bibliography

[1] J. Bell, "Understand the autonomic manager concept." http://www.ibm.com/developerworks/library/ac-amconcept, Feb. 2004. Accessed: 21/03/2013.

[2] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[3] D. T. Maxwell and P. D, "An Overview of The Joint Warfare System (JWARS)," 2000.

[4] A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, C. Cioffi, J. Closier, R. G. Diaz, G. Kuznetsov, Y. Y. Li, R. Nandakumar, S. Paterson, R. Santinelli, A. C. Smith, M. S. Miguelez, and S. G. Jimenez, "DIRAC: a community grid solution," *Journal of Physics: Conference Series*, vol. 119, no. 6, p. 062048, 2008.

[5] M. Welsh, "The Staged Event-Driven Architecture for Highly-Concurrent Server Applications." http://www.eecs.harvard.edu/~mdw/papers/quals-seda.pdf. Accessed: 21/03/2013.

[6] "The GigaSpaces Runtime Environment website." http://wiki.gigaspaces.com/wiki/display/XAP91/The+Runtime+Environment. Accessed: 21/03/2013.

[7] C. Ballinger, "The Teradata Scalability Story." http://www.teradata.com/white-papers/The-Teradata-Database-Scalability-Story-eb3031/. Accessed: 21/03/2013.

[8] "Hadoop tutorial website." http://shiva-dasharathi.appspot.com/tuts/hadoop.html. Accessed: 21/03/2013.

[9] P. Asadzadeh, R. Buyya, C. L. Kei, D. Nayar, and S. Venugopal, "Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies," *CoRR*, vol. cs.DC/0407001, 2004.

[10] Microsoft company, "http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns944/whitepaper_c11-711496.html," last access 14 April, 2013.

[11] "Architecture of the Eucalyptus cloud." http://en.wikipedia.org/wiki/File:Eucalyptus_Platform_Architecture,_February_2013.jpg. Accessed: 21/03/2013.

[12] "Architecture of the OpenStack cloud." http://docs.openstack.org/trunk/openstack-compute/starter/content/Components_of_OpenStack-Compute-d1e166.html. Accessed: 21/03/2013.

[13] A. Oprescu and T. Kielmann, "Bag-of-Tasks Scheduling under Budget Constraints," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 351–359, 2010.

[14] M. Kvassay, L. Hluchý, S. Dlugolinský, M. Laclavík, B. Schneider, H. Bracker, A. Tavčar, M. Gams, D. Król, M. Wrzeszcz, and J. Kitowski, "An integrated approach to mission analysis and mission rehearsal," in *Proceedings of the Winter Simulation Conference*, WSC '12, pp. 362:1–362:2, Winter Simulation Conference, 2012.

[15] Experiment Design and Analysis Reference, "http://www.weibull.com/doewebcontents.htm," last access 14 April, 2013.

[16] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, "An Updated Forecast of Worldwide Information Growth Through 2011," May 2008.

[17] S. Mills, S. Lucas, L. Irakliotis, M. Rappa, T. Carlson, and B. Perlowitz, "DEMYSTIFYING BIG DATA: A Practical Guide to Transforming the Business of Government," tech. rep., 2012.

[18] T. Hey, S. Tansley, and K. Tolle, eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, Washington: Microsoft Research, 2009.

[19] X. Zhu and I. Davidson, *Knowledge Discovery and Data Mining: Challenges and Realities*. Hershey, PA, USA: IGI Publishing, 2007.

[20] D. Kallfass and T. Schlaak, "NATO MSG-088 case study results to demonstrate the benefit of using data farming for military decision support," in *Proceedings of the Winter Simulation Conference*, WSC '12, pp. 221:1–221:12, Winter Simulation Conference, 2012.

[21] A. Brandstein and G. Horne, *Data Farming: A Meta-Technique for Research in the 21st Century*. Marine Corps Combat Development Command Publication, Quantico, Virginia, 1998.

[22] G. E. Horne and K.-P. Schwierz, "Data farming around the world overview," in *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pp. 1442–1447, Winter Simulation Conference, 2008.

[23] T. Meyer and S. Johnson, *Visualization for Data Farming: A Survey of Methods*. Marine Corps Combat Development Command, United States Marine Corps Project Albert. Quantico, Virginia, 2001.

[24] A. Dean and D. Voss, *Design and Analysis of Experiments*. Springer Texts in Statistics, Spring-Verlag, 1999.

[25] C. Engelmann and A. Geist, "Super-Scalable Algorithms for Computing on 100,000 Processors," in *Proceedings of the 5th International Conference on Computational Science*, pp. 313–321, Springer, 2005.

[26] A. Maier, "Keynote: Autonomic Computing Initiative," in *ARCS* (C. Muller-Schloer, T. Ungerer, and B. Bauer, eds.), vol. 2981 of *Lecture Notes in Computer Science*, p. 3, Springer, 2004.

[27] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 289–302, Mar. 2007.

[28] M. Isard, "Autopilot: automatic data center management," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 60–67, Apr. 2007.

[29] I. T. Foster, "The Grid: Beyond the Hype," in *GCC* (H. Jin, Y. Pan, N. Xiao, and J. Sun, eds.), vol. 3251 of *Lecture Notes in Computer Science*, p. 1, Springer, 2004.

[30] E. Simmon and R. Bohn, "An Overview of the NIST Cloud Computing Program and Reference Architecture," in *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment* (J. Stjepandic, G. Rock, and C. Bil, eds.), pp. 1119–1129, Springer London, 2013.

[31] A. Oprescu and T. Kielmann, "Bag-of-Tasks Scheduling under Budget Constraints," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 351–359, 2010.

[32] J. Varia and S. Mathew, "Amazon Web Services." `http://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf`. Accessed: 21/03/2013.

[33] M. Bubak, T. Gubala, M. Malawski, B. Balis, W. Funika, T. Bartynski, E. Ciepiela, D. Harezlak, M. Kasztelnik, J. Kocot, D. Krol, P. Nowakowski, M. Pelczar, J. Wach, M. Assel, and A. Tirado-Ramos, "Virtual Laboratory for Development and Execution of Biomedical Collaborative Applications," in *Proceedings of the 2008 21st IEEE International Symposium on Computer-Based Medical Systems*, CBMS '08, (Washington, DC, USA), pp. 373–378, IEEE Computer Society, 2008.

[34] T. Gubala, B. Balis, M. Malawski, M. Kasztelnik, P. Nowakowski, M. Assel, D. Harezlak, T. Bartynski, J. Kocot, E. Ciepiela, D. Krol, J. Wach, M. Pelczar, W. Funika, and B. Marian, "ViroLab Virtual Laboratory," in *Cracow'07 Grid Workshop : October, 2007, Krakow, Poland*, pp. 35–40, 2007.

[35] P. Nowakowski, D. Harezlak, and M. Bubak, "A New Approach to Development and Execution of Interactive Applications on the Grid," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, (Washington, DC, USA), pp. 681–686, IEEE Computer Society, 2008.

[36] M. Bubak, T. Gubala, M. Malawski, B. Balis, W. Funika, T. Bartynski, E. Ciepiela, D. Harezlak, M. Kasztelnik, J. Kocot, D. Krol, P. Nowakowski, M. Pelczar, and M. Assel, "A platform for collaborative e-science applications," in *Proceedings of 2nd ACC Cyfronet AGH users' conference, Zakopane, 2009*, p. 36, ACC Cyfronet AGH, 2009.

[37] W. Funika, D. Harezlak, D. Krol, and M. Bubak, "Environment for Collaborative Development and Execution of Virtual Laboratory Applications," in *Proceedings of the 8th International Conference on Computational Science, Part III*, ICCS '08, (Berlin, Heidelberg), pp. 446–455, Springer-Verlag, 2008.

[38] W. Funika, D. Harezlak, D. Krol, P. Pegiel, and M. Bubak, "User interfaces of the Virolab Virtual Laboratory," in *Cracow'07 Grid Workshop : October, 2007*, pp. 47–52, 2008.

[39] J. Meizner, M. Malawski, E. Ciepiela, M. Kasztelnik, D. Harezlak, P. Nowakowski, D. Krol, T. Gubala, W. Funika, M. Bubak, T. Mikolajczyk, P. Plaszczak, K. Wilk, and M. Assel, "ViroLab Security and Virtual Organization Infrastructure," in *Advanced Parallel Processing Technologies* (Y. Dou, R. Gruber, and J. Joller, eds.), vol. 5737 of *Lecture Notes in Computer Science*, pp. 230–245, Springer Berlin Heidelberg, 2009.

[40] S. Ambroszkiewicz, *SOA Infrastructure Tools: Concepts and Methods*. Poznań University of Economics Press, 2010.

[41] K. Skalkowski, J. Sendor, M. Pastuszko, B. Puzon, J. Fibinger, D. Krol, W. Funika, B. Kryza, R. Slota, and J. Kitowski, "SOA-based support for dynamic creation and monitoring of virtual organization," in *SOA infrastucture tools concepts and methods* (e. a. S. Ambroszkiewicz, ed.), pp. 371–374, Poznan University of Economics Press, 2010.

[42] W. Funika, P. Pegiel, P. Godowski, and D. Krol, "Semantic-oriented performance monitoring of distributed applications," in *KU KDM 2010 : Third ACC Cyfronet AGH user's conference : Zakopane March, 2010*, pp. 33–34, 2010.

[43] W. Funika, P. Godowski, P. Pegiel, and D. Krol, "Semantic-oriented performance monitoring of distributed applications," *Computing and Informatics*, vol. 31, no. 2, pp. 427–446, 2012.

[44] D. Krol and W. Funika, "Semantic-based SLA-oriented performance monitoring in the ProActive environment," in *Cracow'09 Grid Workshop : October, 2009, Krakow, Poland*, pp. 151–157, 2010.

[45] D. Krol, W. Funika, R. Slota, and J. Kitowski, "SLA-based data storage-oriented semi-automatic management of distributed applications," in *KU KDM 2010 : Third ACC Cyfronet AGH user's conference : Zakopane March, 2010*, p. 39, 2010.

[46] D. Krol, W. Funika, R. Slota, and J. Kitowski, "SLA-Oriented Semi-Automatic Management of Data Storage and Applications in Distributed Environment," *Computer Science*, vol. 11, no. 1, 2010.

[47] PL-Grid project, "`http://www.plgrid.pl/en`," last access 14 April, 2013.

[48] R. Slota, D. Krol, K. Skalkowski, B. Kryza, D. Nikolow, M. Orzechowski, and J. Kitowski, "A Toolkit for Storage QoS Provisioning for Data-Intensive Applications," in *Building a National Distributed e-Infrastructure PL-Grid* (M. Bubak, T. Szepieniec, and K. Wiatr, eds.), vol. 7136 of *Lecture Notes in Computer Science*, pp. 157–170, Springer Berlin Heidelberg, 2012.

[49] PLGrid Plus project, "`http://www.plgrid.pl/en#section-1t`," last access 14 April, 2013.

[50] R. Slota, D. Krol, K. Skalkowski, M. Orzechowski, D. Nikolow, B. Kryza, M. Wrzeszcz, and J. Kitowski, "A Toolkit for Storage QoS Provisioning for Data-Intensive Applications," *Computer Science*, vol. 13, no. 1, 2012.

[51] R. Slota, D. Nikolow, J. Kitowski, D. Krol, and B. Kryza, "FiVO/QStorMan Semantic Toolkit for Supporting Data-Intensive Applications in Distributed Environments," *Computing and Informatics*, vol. 31, no. 5, pp. 1003–1024, 2012.

[52] K. Skalkowski, R. Slota, D. Krol, M. Orzechowski, B. Kryza, and J. Kitowski, "Towards scalable, semantic-based virtualized storage resources provisioning," in *KU KDM 2012 : fifth ACC Cyfronet AGH user's conference*, pp. 76–77, 2012.

[53] R. Slota, D. Krol, K. Skalkowski, B. Kryza, D. Nikolow, and J. Kitowski, "FiVO/QStorMan: toolkit for supporting data-oriented applications in PL-Grid," in *KU KDM 2011 : fourth ACC Cyfronet AGH users' conference : Zakopane, March, 2011*, p. 68, ACK Cyfronet AGH, 2011.

[54] D. Krol, R. Slota, B. Kryza, D. Nikolow, W. Funika, and J. Kitowski, "Policy Driven Data Management in PL-Grid Virtual Organizations," in *Remote Instrumentation for eScience and Related Aspects* (F. Davoli, M. Lawenda, N. Meyer, R. Pugliese, J. Weglarz, and S. Zappatore, eds.), pp. 257–266, Springer New York, 2012.

[55] D. Krol, A. Chrabaszcz, R. Slota, and J. Kitowski, "Evaluation of QStorMan dynamic storage provisioning strategies in PL-Grid," in *Cracow'12 Grid Workshop : October, 2012, Krakow, Poland*, pp. 81–82, 2012.

[56] D. Krol, B. Kryza, K. Skalkowski, D. Nikolow, R. Slota, and J. Kitowski, "QoS provisioning for data-oriented applications in PL-Grid," in *Cracow'10 Grid Workshop : October, 2010, Krakow, Poland*, pp. 149–150, 2010.

[57] K. Skalkowski, R. Slota, D. Krol, and J. Kitowski, "Qos-based storage resources provisioning for grid applications," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 713 – 727, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[58] D. Krol, B. Kryza, M. Wrzeszcz, L. Dutka, and J. Kitowski, "Elastic Infrastructure for Interactive Data Farming Experiments," *Procedia Computer Science*, vol. 9, no. 0, pp. 206 – 215, 2012. Proceedings of the International Conference on Computational Science, {ICCS} 2012.

[59] D. Krol, M. Wrzeszcz, B. Kryza, L. Dutka, and J. Kitowski, "Scalarm: massively self-scalable platform for data farming," in *Cracow'12 Grid Workshop : October, 2012, Krakow, Poland* (K. W. Marian Bubak, Michal Turala, ed.), pp. 53–54, Academic Computer Centre CYFRONET AGH, 2012.

[60] S. Upton, "Users Guide: OldMcData, the Data Farmer, Version 1.1." `http://harvest.nps.edu/software.html`. Accessed: 21/03/2013.

[61] "SEED Center for Data Farming website." `http://harvest.nps.edu`. Accessed: 21/03/2013.

[62] S. Upton, "XStudy application website." `http://harvest.nps.edu/software.html`. Accessed: 21/03/2013.

[63] R. P. Bruin, T. O. H. White, A. M. Walker, K. F. Austen, M. T. Dove, R. P. Tyer, P. A. Couch, I. T. Todorov, and M. O. Blanchard, "Job submission to grid computing environments," in *Proceedings of the UK e-Science All Hands Meeting 2006*, (Nottingham, UK), pp. 426–432, 2006.

[64] I. T. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," *J. Comput. Sci. Technol.*, vol. 21, no. 4, pp. 513–520, 2006.

[65] G. F. S. III and G. A. McIntyre, "JWARS: the joint warfare system (JWARS): a modeling and analysis tool for the defense department," in *Winter Simulation Conference*, pp. 691–696, 2001.

[66] M. Scheutz, P. Schermerhorn, R. Connaughton, and A. Dingler, "SWAGES-An Extendable Distributed Experimentation System for Large-Scale Agent-Based Alife Simulations."

[67] N. Brook, A. Bogdanchikov, A. Buckley, J. Closier, U. Egede, M. Frank, D. Galli, M. Gandelman, V. Garonne, C. Gaspar, R. G. Diaz, K. Harrison, E. van Herwijnen, A. Khan, S. Klous, I. Korolko, G. Kuznetsov, F. Loverre, U. Marconi, J. P. Palacios, G. N. Patrick, A. Pickford, S. Ponce, V. Romanovski, J. J. Saborido, M. Schmelling, A. Soroko, A. Tsaregorodtsev, V. Vagnoni, and A. Washbrook, "DIRAC - Distributed Infrastructure with Remote Agent Control," *CoRR*, vol. cs.DC/0306060, 2003.

[68] A. A. Alves *et al.*, "The LHCb Detector at the LHC," *JINST*, vol. 3, p. S08005, 2008.

[69] M. D. Welsh, *An architecture for highly concurrent, well-conditioned internet services*. PhD thesis, 2002. AAI3082454.

[70] M. Welsh, D. E. Culler, and E. A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *SOSP*, pp. 230–243, 2001.

[71] "SwiftMQ website." `http://www.swiftmq.com/`. Accessed: 21/03/2013.

[72] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Y. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," in *ASPLOS* (L. Rudolph and A. Gupta, eds.), pp. 190–201, ACM Press, 2000.

[73] "Understanding the Cost of Data Center Downtime: An Analysis of the Financial Impact on Infrastructure Vulnerability." `http://emersonnetworkpower.com/en-US/Brands/Liebert/Documents/White%20Papers/data-center-uptime_24661-R05-11.pdf`. Accessed: 21/03/2013.

[74] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," 2004.

[75] T. White, *Hadoop: The Definitive Guide*. O'Reilly, first edition ed., june 2009.

[76] R. C. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," *BMC Bioinformatics*, vol. 11, no. S-12, p. S1, 2010.

[77] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1 –10, May 2010.

[78] A. S. Foundation, "The zookeeper project website." `http://zookeeper.apache.org/doc/current/zookeeperOver.html`. Accessed: 21/03/2013.

[79] K. V. Shvachko, "Apache Hadoop: The Scalability Update." `https://www.usenix.org/publications/login/june-2011-volume-36-number-3/apache-hadoop-scalability-update`, 2011. Accessed: 21/03/2013.

[80] C. Kesselman and I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[81] A. Streit, S. Bergmann, R. Breu, J. M. Daivandy, B. Demuth, A. Giesler, B. Hagemeier, S. Holl, V. Huber, D. Mallmann, A. S. Memon, M. S. Memon, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and T. Lippert, "UNICORE 6 - A European Grid Technology," in *High Performance Computing Workshop* (W. Gentzsch, L. Grandinetti, and G. R. Joubert, eds.), vol. 18 of *Advances in Parallel Computing*, pp. 157–173, IOS Press, 2008.

[82] B. Bosak, J. Konczak, K. Kurowski, M. Mamonski, and T. Piontek, "Highly Integrated Environment for Parallel Application Development Using QosCosGrid Middleware," in *PL-Grid* (M. Bubak, T. Szepieniec, and K. Wiatr, eds.), vol. 7136 of *Lecture Notes in Computer Science*, pp. 182–190, Springer, 2012.

[83] S. Bounanos and M. Fleury, "Gb Ethernet Protocols for Clusters: An Open-MPI, TIPC, GAMMA Case Study.," in *PARCO* (C. H. Bischof, H. M. Bucker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, eds.), vol. 15 of *Advances in Parallel Computing*, pp. 397–404, IOS Press, 2007.

[84] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici, *Grid Computing: Software Environments and Tools*, ch. Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[85] Amazon, "Amazon Elastic Compute Cloud." Online http://aws.amazon.com/ec2/, 2010.

[86] AWS Auto Scaling feature, "`http://aws.amazon.com/autoscaling/`," last access 14 April, 2013.

[87] Windows Azure Autoscaling Application Block, "`http://msdn.microsoft.com/en-us/library/hh680892(v=pandp.50).aspx`," last access 14 April, 2013.

[88] S. Krishnan, *Programming Windows Azure - Programming the Microsoft Cloud*. O'Reilly, 2010.

[89] M. Malawski, M. Kuzniar, P. Wojcik, and M. Bubak, "How to Use Google App Engine for Free Computing," *IEEE Internet Computing*, vol. 17, no. 1, pp. 50–59, 2013.

[90] N. Mirajkar, M. Barde, H. Kamble, R. Athale, and K. Singh, "Implementation of Private Cloud using Eucalyptus and an open source Operating System," *CoRR*, vol. abs/1207.3037, 2012.

[91] I. C. Plasencia, E. F. del Castillo, S. Heinemeyer, A. Lopez-Garcia, and F. v. d. Pahlen, "Phenomenology Tools on Cloud Infrastructures using OpenStack," *CoRR*, vol. abs/1212.4784, 2012.

[92] S. Saini, S. Heistand, H. Jin, J. Chang, R. Hood, P. Mehrotra, and R. Biswas, "An Application-based Performance Evaluation of NASA's Nebula Cloud Computing Platform.," in *HPCC-ICESS* (G. Min, J. Hu, L. C. Liu, L. T. Yang, S. Seelam, and L. Lefevre, eds.), pp. 336–343, IEEE Computer Society, 2012.

[93] "Rackspace Cloud File website." `http://www.rackspace.com/cloud/files/`. Accessed: 21/03/2013.

[94] R. Cushing, S. Koulouzis, A. Belloum, and M. Bubak, "Applying workflow as a service paradigm to application farming," *Concurrency and Computation: Practice and Experience*, 2013.

[95] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti, "Cloud Federations in Contrail," in *Euro-Par 2011: Parallel Processing Workshops* (M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. Scott, J. Traff, G. VallĂŠe, and J. Weidendorfer, eds.), vol. 7155 of *Lecture Notes in Computer Science*, pp. 159–168, Springer Berlin Heidelberg, 2012.

[96] G. Pierre and C. Stratan, "ConPaaS: A Platform for Hosting Elastic Cloud Applications," *Internet Computing, IEEE*, vol. 16, no. 5, pp. 88–92, 2012.

[97] M. Gaudard, P. Ramsey, and M. Stephens, "Interactive Data Mining and Design of Experiments: the JMP Partition and Custom Design Platforms." `http://www.jmp.com/software/whitepapers/pdfs/372455_interactive_datamining.pdf`. Accessed: 21/03/2013.

[98] R Development Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[99] M. Kircher and P. Jain, "The Three-Tier Architecture Pattern Language Design Fest.," in *EuroPLoP* (A. Ruping, J. Eckstein, and C. Schwanninger, eds.), pp. 575–580, UVK - Universitaetsverlag Konstanz, 2001.

[100] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTRs, Aug. 2005.

[101] N. Shalom, "The Scalability Revolution: From Dead End to Open Road." `http://wiki.gigaspaces.com/wiki/download/attachments/1835009/FromDeadEndToOpenRoad.pdf`. Accessed: 21/03/2013.

[102] O. Kremien, "Scalability in distributed systems, parallel systems and supercomputers.," in *HPCN Europe* (L. O. Hertzberger and G. Serazzi, eds.), vol. 919 of *Lecture Notes in Computer Science*, pp. 532–541, Springer, 1995.

[103] V. P. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures.," *J. Parallel Distrib. Comput.*, vol. 22, no. 3, pp. 379–391, 1994.

[104] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[105] Y. Shi, "Reevaluating Amdahl's Law and Gustafson's Law." `http://spartan.cis.temple.edu/shi/public_html/docs/amdahl/amdahl.html`, 1996. Accessed: 21/03/2013.

[106] P. Jogalekar and C. M. Woodside, "Evaluating the Scalability of Distributed Systems.," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 6, pp. 589–603, 2000.

[107] T. Horikawa and A. Fukuda, "A method for analysis and solution of scalability bottleneck in DBMS," in *Proceedings of the 2010 Symposium on Information and Communication Technology*, SoICT '10, (New York, NY, USA), pp. 139–146, ACM, 2010.

[108] T. Brisco, "DNS Support for Load Balancing." RFC 1794 (Informational), April 1995.

[109] R. Brachman and H. Levesque, *Knowledge Representation and Reasoning*. Amsterdam: Morgan Kaufmann, 2004.

[110] N. Dunstan, "Generating domain-specific web-based expert systems.," *Expert Syst. Appl.*, vol. 35, no. 3, pp. 686–690, 2008.

[111] K. Banker, *MongoDB in action*. Manning Pubs Co Series, Manning Publications Company, 2011.

[112] D. Hildebrand, P. Honeyman, and D. Hildebrand, "pNFS and Linux: Working Towards a Heterogeneous Future," in *In 8th LCI International Conference on High-Performance Cluster Computing (Lake Tahoe*, 2007.

[113] Using a Service Locator pattern, "`http://martinfowler.com/articles/injection.html#UsingAServiceLocator`," last access 14 April, 2013.

[114] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2007.

[115] M. Laclavik, S. Dlugolinsky, M. Seleng, M. Kvassay, B. Schneider, H. Bracker, M. Wrzeszcz, J. Kitowski, and L. Hluchy, "Agent-Based Simulation Platform Evaluation in the Context of Human Behavior Modeling," in *AAMAS Workshops* (F. Dechesne, H. Hattori, A. ter Mors, J. M. Such, D. Weyns, and F. Dignum, eds.), vol. 7068 of *Lecture Notes in Computer Science*, pp. 396–410, Springer, 2011.

[116] S. Dlugolinsky, M. Kvassay, L. Hluchy, M. Wrzeszcz, D. Krol, and J. Kitowski, "Using parallelization for simulation of human behaviour," in *Proceedings of the 7th International Workshop on Grid Computing for Complex Problems*, GCCP 2011, (Bratislava, Institute of Informatics SAS), pp. 258–265, 2011.

[117] A. Tavcar, M. Gams, M. Kvassay, M. Laclavík, L. Hluchy, B. Schneider, and H. Bracker, "Graph-based analysis of data from human behaviour simulations," in *Applied Machine Intelligence and Informatics (SAMI), 2012 IEEE 10th International Symposium on*, pp. 421–426, 2012.

[118] B. Kryza, D. Krol, M. Wrzeszcz, L. Dutka, and J. Kitowski, "Interactive cloud data farming environment for military mission planning support," *Computer Science : rocznik Akademii Gorniczo-Hutniczej imienia Stanislawa Staszica w Krakowie*, vol. 13, no. 3, pp. 89–100, 2012.

[119] T. Cioppa, *Efficient Nearly Orthogonal and Space-Filling Experimental Designs for High-Dimensional Complex Models*. Storming Media, 2002.

[120] D. Krol, M. Wrzeszcz, B. Kryza, L. Dutka, and J. Kitowski, "Massively Scalable Platform for Data Farming Supporting Heterogeneous Infrastructure," in *The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization*, IARIA Cloud Computing 2013, (Valencia, Spain), pp. 144–149, 2013.

[121] D. Krol, R. Slota, and W. Funika, "Behaviour-inspired Data Management in the Cloud," in *Proc. of CLOUD COMPUTING 2010 The First International Conference on Cloud Computing, GRIDs, and Virtualization*, IARIA CLOUD COMPUTING 2010, pp. 98–103, IARIA, 2010.

[122] D. Krol, R. Slota, and W. Funika, "Behaviour-inspired Data Management in the Cloud," *International Journal on Advances in Intelligent Systems*, vol. 4, no. 3 & 4, pp. 256–267, 2011.

[123] D. Krol and J. Kitowski, "Distributed Storage Support in Private Clouds Based on Static Scheduling Algorithms," in *Proc. of CLOUD COMPUTING 2011 The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, IARIA CLOUD COMPUTING 2011, pp. 141–146, IARIA, 2011.

Index