Joseph Fourier University, ENSIMAG
Master of Science in Informatics at Grenoble
Distributed, Embedded, Mobile, Interactive and Parallel Systems

---

Elasticity in Cloud Computing

Realized by :
Petr Sobeslavsky

Date of Defence :
June 23, 2011

Team Sardes, Inria Rhône-Alpes.

Supervisors :
Noël De Palma
Fabienne Boyer

In Collaboration with :
Bruno Dillenseger (Orange Labs)

JURY :

Florence Maraninchi	, Permanent jury member
Christine Verdier	, Permanent jury member
Marie-Christine Rousset	, Permanent jury member
Jean-Marc Vincent	, External examiner
Noel De Palma	, Supervisor
Fabienne Boyer	, Supervisor

Acknowledgements

I would firstly like to thank my supervisors Noel de Palma and Fabienne Boyer for their guidance and advice throughout the whole duration of my internship. I would also like to thank other colleagues from the SARDES team for their comments and support.

My thanks must also go to Bruno Dillenseger for his time and assistance with the implementation part of the project.

Abstract

One of yet unresolved challenges of cloud computing is the problem of making an application elastic, which consists in making it automatically adjust to variations in load without the need of intervention of a human administrator and without the need to change its code.

In this project we first identified several design issues that have to be addressed when making an application elastic: defining the granularity of the elastic components in the application, handling their interconnections and controlling the elasticity at execution time. We studied the approach of autonomic computing and proposed an architecture of an elastic application manager with a well defined separation between general concepts and application-specific parts.

We used a load injection application (Clif) as a use case. We studied the architecture of the application and the Fractal component model it is based on and implemented an elastic manager. In an experiment with a performance evaluation of a web application we have shown that the approach is feasible and working.

Keywords: Cloud computing, autonomic computing, elasticity, Fractal, Clif

Résumé

Un des défis du cloud computing consiste à faire une application élastique, c'est-à-dire à ajuster l'application aux variations de charge sans intérvention d'un administrateur humain et sans changement du code.

Dans ce projet nous avons d'abord idéntifié plusieurs challenges dans la conception d'une application élastique : détermination de la granularité des composants élastiques, gestion des interconnexions des composants et contrôle de l'élasticité pendant l'éxécution. Nous avons étudié l'approche de l'informatique autonome et nous avons proposé une architecture d'un controlleur d'élasticité avec une séparation bien definie entre les concepts géneraux et la partie spécifique pour l'application.

Comme cas d'étude nous avons utilisé un injecteur de charge (Clif). Nous avons étudié l'architecture de l'application et le modêle de composants Fractal. Nous avons experimenté avec une évaluation de la performance d'une application web et nous avons montré la faisabilité et l'applicabilité de notre approche.

Mots-clés : Cloud computing, informatique autonome, élasticité, Fractal, Clif

Contents

Chapter I

Introduction

1 Context of Work

The work presented in this report was realized as part of project SARDES [1]. SARDES is a project and a research team of the LIG laboratory located at INRIA Grenoble-Rhône-Alpes.

The research domain of SARDES is distributed systems – systems composed of spatially distributed machines interconnected with a network. The main research interest is then the construction of MultiScale Open Systems – distributed systems that can scale from resource-constrained embedded systems to cloud-based systems.

Such systems are known for being highly complex, dynamic and heterogeneous. It is often impossible for an administrator to keep track of their current state and do their administration efficiently. The aim is to develop self-managed software infrastructures which would be able to operate autonomously without the need for an external administrator.

Within SARDES, the subject of work was to provide a solution to manage the scalability of the applications deployed in cloud environment through a particular technique called elasticity. The work was done in collaboration with Orange Labs (Bruno Dillenseger).

2 Cloud Computing

Cloud computing platforms have received a great deal of attention in the business world in recent years. The main motivation for companies to consider transferring their existing systems or creating new ones on top of a cloud platform is the flexibility they promise to provide and their pricing model.

The fundamental difference between a classical approach with a fixed infrastructure and cloud computing can be illustrated on an example of a startup company launching a new service on the web.

With the fixed infrastructure approach, the company first has to estimate the amount of hardware it will need to provide the service with a required level of quality to the estimated number of customers. Then it has to acquire the hardware (by buying or by renting), install the application and start providing the service. However as the number of clients is often difficult to predict and changes significantly in time (e.g.

1. SARDES: System Architecture for Reflective Distributed Environments

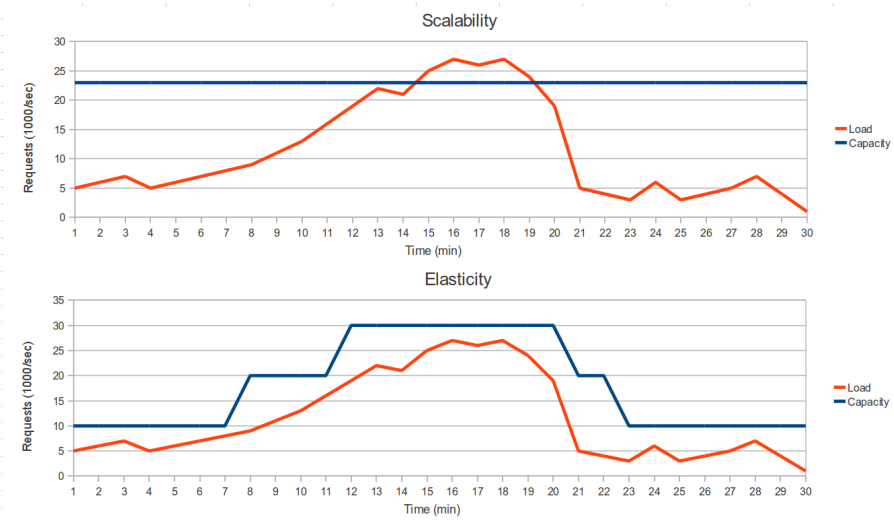

Figure I.1 – Comparison of classical scalability and elasticity approaches: In the first case the system has been enlarged to a capacity corresponding to the expected maximum load. For most of the time its capacity is not used, while at the peak the load is higher than the expected value. In the second case, the system reacts dynamically to the change in the load, adapting as needed and leaving just a small part of its capacity unused.

a publication of an article about the company on a highly visited news server can multiply the number of visitors in a short time period), the company risks that it will either buy more hardware than necessary and pay unnecessary costs or on the other hand that the hardware will not be sufficient enough, the quality of the service will be low and will discourage potential clients.

Cloud computing claims to provide a solution to this problem: Instead of buying a hardware and building its own infrastructure, the company rents capacity from a cloud provider and only pays for the time it is actually used. The cloud computing provider runs huge data centers with hundreds of servers and is thus able to pay lower prices for the hardware and lower operating costs. According to a study of Armbrust et al. [8], the saving on costs can be of factor 5 to 7. Instead of paying a high fixed cost of installation and subsequent operating costs, the company would only pay the operating costs.

Another advantage of the cloud computing is in the elasticity it can provide to applications. In a huge data center, new resources can be allocated and assigned to clients in a short time. The startup company could flexibly allocate new resources when the number of visitors augments and later release these resources when they are not needed anymore. This way it could prevent the deterioration in the quality of service and save money on not paying for non-used infrastructure.

3 Application Elasticity

The variations in load pose a specific challenge on applications in distributed environments. In the previous section we presented an example with a web application,

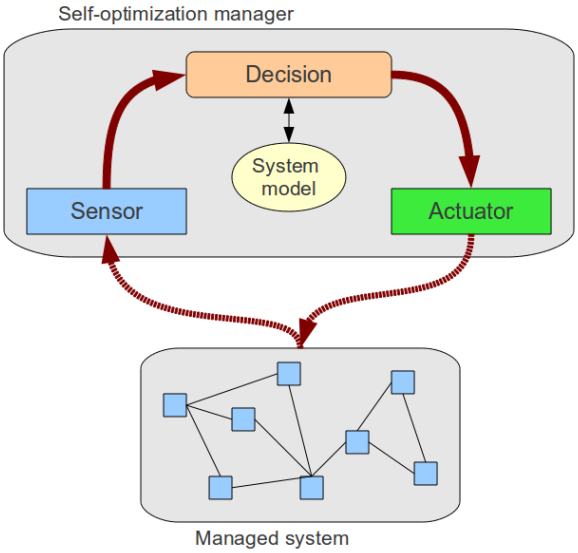

Figure I.2 – Self-configuration control loop

however such variations can happen in generally any kind of application. A traditional solution to this problem has been to ensure **scalability** – the ability of the system to be enlarged to a size which was expected to accommodate a future growth. The disadvantage of such an approach is that the size of the system has to be estimated in advance. If the growth in the load does not correspond to the estimation, the assigned capacity is not used effectively. Another disadvantage is that scaling up implies extending the physical resources, and it may be difficult to scale down after a scale up.

A possible solution to the problem is to ensure **elasticity**. Elasticity aims to solve the problem in an opposite direction - instead of setting the target physical size of the system in advance, the system dynamically reacts to actual load by adding new virtual resources. When the load on a component increases over a given limit, new instance of the component is added to accommodate the growth. If on the other hand the load decreases, unnecessary instance can be removed. Figure I.1 illustrates the difference between the two approaches.

4 Autonomic Computing

The presented problem falls into the category of problems of autonomic computing [14]. In general, **autonomic computing** can be explained as an effort to develop self-managed complex software systems that would have the following characteristics:

- *Self-configuration* – automatic configuration of components
- *Self-healing* – automatic discovery and correction of faults
- *Self-optimization* – automatic provisioning of resources
- *Self-protection* – automatic identification and protection from attacks

The problem of application elasticity is thus a self-optimization problem.

Autonomic computing makes use of autonomic managers which implement feedback control loops. These loops regulate and optimize the behavior of the managed system. Figure I.2 presents such a loop in a self-optimization manager. The self-optimization manager is a component external to the managed system. It consists of a sensor which periodically checks the actual state of the system, a decision module which decides whether an action has to be taken and an actuator which modifies the system. The manager does not have detailed knowledge of the system – it only works with a simplified model.

5 Subject of Work

Our main research objective was to provide a general solution for the management of application elasticity in the context of cloud environment. Particularly, we aimed to find a solution which would:

- Provide reactive ways of scaling up and down
- Minimize the impact of implementation of elasticity on the way the applications are programmed

In order to show that the concept is practically feasible, the goal was to implement an elastic manager, which would be executed as a separate application and make an existing application elastic without modifying its source code. We used a load injector Clif as a use case.

Chapter II

State of the Art

This chapter first overviews several existing cloud computing platforms and describes their approach to application elasticity. In the second part, several research projects dealing with the issue are presented. The third part summarizes our observations.

1 Commercial Platforms

Existing commercial platforms can be divided into three classes according to the layer at which the platform interfaces with the application (see Figure II.1):

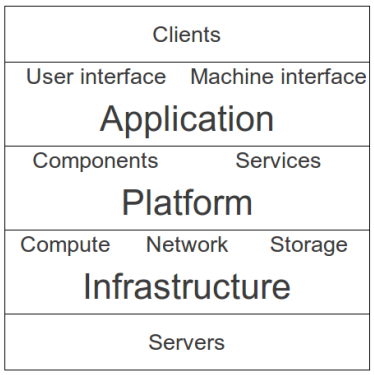

Figure II.1 – Cloud computing layers

- *Infrastructure as a Service (IaaS)* – the service provider provides basic infrastructure, typically just empty virtual machines. A client has to install his own operating system and applications.
- *Platform as a Service (PaaS)* – the service platform provides a platform on which client's applications have to be developed and executed. The client does not care about the administration of the underlying hardware and operating system, he only has to provide applications.
- *Software as a Service (SaaS)* – the whole stack is provided by the service provider. A client is not requested to do any administrative tasks and only pays for usage.

As we can see the services in the lower layers require the clients to handle more administrative tasks, while giving them more freedom in the choice of software and programming models to use. Services in the higher levels make administration easier, but restrict the variety of applications that can be run.

We have selected three commercial cloud computing platforms to illustrate different approaches to cloud computing and their ability to provide elastic services. In the subsequent sections, we always first provide a general overview of a platform and then evaluate its suitability for elastic applications.

Amazon EC2

Amazon EC2 [1] is the only infrastructure as a service in the comparison. A client uploads a virtual image of his system and runs it on Amazon servers. The client either provides the whole virtual machine with the operating system and applications or chooses one of the prepared images.

For each instance, the client chooses the type (which defines the computation capacity in number of CPUs, data storage capacity etc.) and one of the three purchasing options: *On-demand* instances are flexible and are paid by the number of hours they ran. *Reserved* instances guarantee capacity during all the time. *Spot* instances allow the client to set the maximum value he is willing to pay per hour. Depending on the current usage of the cloud, the price per hour is changing. When the price is lower that the threshold, the instance is executed, when it gets higher, it is terminated.

Amazon EC2 provides an auto-scaling mechanism for application instances. It works in the following way: The user defines lower and upper thresholds on CPU, memory or network usage of the application instance and a time period (in the number of minutes) over which the value is evaluated. If a threshold is exceeded, the number of instances is adjusted (new instance started or a running instance terminated). The time of reaction is at least one minute plus the time needed to start a new instance. An Elastic Load Balancer is provided at the TCP layer to distribute the load over the running instances.

Amazon EC2 provides two options for storing data: Elastic Block Store is an unformatted block device with capacity from 1 GB to 1 TB. It can be used as a block device for virtual images or be mounted as a standard block device to store data files. Amazon Simple Storage Service (S3) is a scalable data object storage. User can store data objects of size from 1 byte to 5 GB identified by a unique key and retrieve them afterwards. It provides REST and SOAP based interfaces.

There are two database options: SimpleDB is a scalable non-relational data store. It stores items described as attribute-value pairs organized in sets called domains. It provides a simple API to modify the attributes and values and to retrieve data using a Select command. The Select only works on one domain. Relational Database Service (RDS) is a MySQL 5.1 compatible database. For the database instance, the client chooses the instance type (sets the computation capacity and storage capacity). RDS can create Read replicas of a database instance.

Amazon EC2 is an IaaS service, thus it enables the client to run applications based on an arbitrary platform with any specific settings. Applications can be scaled automatically, the unit of scalability is the whole instance and the reaction time is in the order of minutes.

Microsoft Windows Azure

Azure [5] is a platform as a service offering from Microsoft. It is based on already established Microsoft products – the Windows operating system, the .NET platform and the SQL server – and aims to enable easy transfer of existing applications to the cloud.

The cloud provides the operating system, runtime environment and distributed database. The client is only required to upload and configure its application. The application is called a service and consists of one or several roles. Each role can be run in one or several instances. There are two types of roles – web role is optimized for web application programming supported by IIS 7 and ASP.NET, worker role is intended for general application development, it might offer background processing for a web role.

Services are developed, compiled and packaged using Windows Azure Tools, which is an extension to Microsoft's standard development tools (Visual Studio or Visual Web Developer). A developer can use any programming language supported by the .NET platform and use the .NET Framework 3.5 library. The service package is accompanied by a configuration file, which specifies the roles, the number of instances of each role to run and the type of hardware configuration.

It is possible to execute several instances of one role – these instances are automatically grouped into upgrade domains. It is possible to change configuration of instances and restart the instances by upgrade domains (i.e. the operation of start/stop affects the whole domain). This can be done on-the-fly as long as there are no new roles added to the application. A client can carry out these changes either via a web interface or a REST API. There is a diagnostic API which a client can use to get runtime information about the instances (number of requests, processing time, cpu, disk, memory, network usage,...). However there is no tool to modify the instance settings automatically. It is only possible to setup e-mail or instant messaging notifications and do the changes manually using the REST API. The billing window on running time of an instance is one hour.

Azure storage services can store binary large objects, queues or tables and provide access via a REST API. Microsoft also offers SQL Azure. It is a relational database based on the SQL Server and provides the same interface, so it is possible to use the whole set of SQL Server features (tables, views, procedures, triggers, ...). The database is replicated in the data center and data recovery in case of failure is assured by the provider. The size of one database is limited to 10 GB, a client account can contain several databases. Microsoft encourages the clients to use scale-out on the databases – a horizontal scaling by dividing data into several databases and accessing them in parallel. This way it can better distribute the database workload in its datacenters, however it is a complex issue for the implementation at the client's side as the whole logic of increasing and reducing the number of databases, distribution of data etc. has to be done by the application.

Azure is based on .NET platform so it is possible for the developer to use the whole range of existing libraries and development tools. It does not provide any built-in support for automatic scaling of applications. It is not possible to change the number of running instances on-the-fly. If the number of instances changes, the whole upgrade domain has to be restarted.

Google App Engine

Google App Engine [4] is a platform as a service offering from Google. It claims to be written as language-independent, however at the current time, only support for Java and Python is provided. The applications are isolated from the running platform and can only access it using the provided API. Java applications can only use a limited subset of the JRE (and they are not allowed to create new threads), Python applications can use frameworks supporting WSGI (such as Django, CherryPy, ...). There are special APIs for the platform functions – URLfetch for HTTP access to the outside world, mail, XMPP, image manipulation, memcache. Access to the file system is read-only.

The App Engine is optimized for Web Applications – an application can be invoked by a HTTP/HTTPS call or as a cron job. There is no support for stand-alone applications. Once invoked, the application has maximum of 30 seconds to complete its execution and return the response.

Java applications are deployed as servlets with a deployment descriptor XML file. Python applications are deployed as a batch of source files with a configuration in a YAML file. A client cannot select hardware for the application instance – all applications run in a uniform environment and are regulated by quotas. There are two types of quotas – billable quotas are set by the user depending on his budget and are application-specific. Fixed quotas are set by the App Engine and cannot be exceeded by any application in the cloud. These quotas cover CPU usage, bandwidth used, number of e-mails sent and the amount of data in the data storage. Furthermore, there are per-minute quotas on CPU time, number of requests and bandwidth. A client cannot change the per-minute quota as it is defined by the App Engine. The number and location of executed instances of the application is determined automatically – it can be seen in the administration console, but cannot be changed by the user.

For storing the data, Google App Engine provides the Data Store – a non-relational, schema less database. It stores items described as property-value pairs in tables. The entity corresponds to an object and the name of the table is the name of the class. There is a simple query language GQL with a SELECT command which only works on one table. There is no support for joining the tables, one-to-many and many-to-many relations can be emulated using a ReferenceProperty mechanism. The entity size is restricted to 1 MB, number of entities affected by a PUT and DELETE commands is restricted to 500.

A blob store is able to store binary objects identified by unique keys of size up to 2 GB.

Google App Engine provides an easy administration of the application, as the execution details are hidden from the user. It is not possible to implement a wide variety of applications. Stand-alone applications are not supported, only a subset of the JRE can be used, there is no other network connection with the outside world than HTTP/HTTPS. The quota system is restrictive and does not allow for high elasticity: the fixed quotas cannot be changed and the per-minute quotas prevent the application from flexibly reacting to short periods of high demand.

2 Research Projects

We have selected several research projects which solve an issue similar to elasticity or which could provide some inspiration in the way they handle the management of a distributed application.

BOINC [7] is an infrastructure for public-service computing. The purpose is for scientist to implement projects which require a big amount of computational resources and for volunteers connected via internet to participate on their computation.

A project is divided into a set of workunits – each workunit represents the inputs to a computation. It has parameters such as compute, memory and storage requirements and a soft deadline for completion. A result is a set of output files for a given workunit. A client registers for a project and connects to the project server. Here it downloads a workunit, computes individually the result and sends it back to the server.

Because the clients are not reliable, can connect or disconnect at any time and may even provide invalid results, BOINC introduces the concept of redundant computing - the same unit is sent to several clients and the results they provide are compared to verify that they are correct.

BOINC in an established platform with more than 300 000 users, which proves that such a system is feasible in a large scale, however the scope of applications is by definition limited to problems where input can be easily divided to small parts and processed independently.

ASKALON [13] is a system used to develop and port scientific applications as workflows in the Austrian Grid project. A user composes Grid workflow applications at a high-level of abstraction using an XML-based language (AGWL). The AGWL representation of a workflow is then given to the middleware services (run-time system) for scheduling and reliable execution. In AGWL, a workflow application is composed from atomic units of work called activities interconnected through control flow and data flow dependencies. Activities are represented at two abstract levels: activity types and activity deployements. An acitivity type is a simplified abstract description of functions or semantics of an activity, whereas an activity deployment refers to an executable or a deployed Web service and describes how they can be accessed and executed on the grid. The control flow constructs include sequences, directed acyclic graphs, for, while, do-while loops and parallel activities. Basic data flow is specified by connecting input and output ports between activities.

ASKALON middleware provides the following set of services: The resource manager is responsible for negotiation, reservation and allocation of resources, as well as automatic deployment of service required to execute grid applications. The enactment engine service targets reliable and fault-tolerant execution of workflows through techniques such as checkpointing, migration, restart, retry and replication. Performance analysis supports automatic instrumentation and bottleneck detection (e.g. excessive synchronization, communication, load imbalance, inefficiency, non-scalability). The performance prediction focuses on estimating execution times of workflow activities through training phase and statistical methods. The scheduler is a service that determines effective mappings of single or multiple workflow applications onto the grid using graph-based heuristics and optimization algorithms on top of the performance prediction and resource manager services.

ASKALON solves the problem of deployment of work-flow applications with a known structured in a known environment. It uses a prediction system which is optimized for scientific applications, where the behavior is predictable and benchmarking runs can be executed at the beginning. The topology of the applications is fixed at the beginning, there is no space for elasticity.

Demberel et al. [11] focuses on an application which uses server resources from a shared computer infrastructure opportunistically. An external controller launches application functions based on a knowledge of what resources are available from the cloud, their cost, and their value to the application through time. The application has to be able to adapt and use the assigned resources as efficiently as possible.

A special case of an application is covered – a server performance measurement tool which runs a set of experiments on servers while setting different configuration parameters and evaluates their performance. Based on the measurements of already processed experiments it estimates the needs of the remaining experiments and uses a greedy heuristics to schedule the ones that would most efficiently use the available resources.

Such an approach only works with applications where it is easy to estimate the future costs of tasks based on the observations of previously run similar tasks.

Lim et al. [15] addresses elastic control for multi-tier application services that allocate and release resources in discrete units, such as virtual server instances of predetermined sizes. It focuses on the elastic control of the storage tier, in which adding or removing a storage node requires rebalancing stored data across the nodes.

The target environment is an elastic guest application hosted on server instances obtained on a pay-as-you-go basis from a cloud provider. The application has defined a Service Level Objective (SLO) that characterizes the target level of acceptable performance, typically a maximum acceptable response time for a web application. The purpose of the elasticity if to grow and shrink the active server instance as needed to meet the SLO under the observed or predicted workload.

The controller process collects the inputs from sensors, analyzes them and drives actuators, which control the application. The actuators operate in a discrete way, i.e. they can only add or remove one whole instance of a resource. The storage tier is a distributed service that runs on a group of service instances provisioned for storage. It exports an API which allows a newly acquired instance to join the group of instances or an arbitrary instance to leave a group. When the number of instances changes, the storage engine redistributes (rebalances) the data.

The horizontal scale controller is responsible for growing or shrinking the number of storage nodes. When the average CPU utilization on instances exceeds a maximum threshold, a new instance is added. When the average CPU utilization gets lower than a minimum threshold, an instance is removed. This way the average CPU utilization is kept in the limits which assure efficient usage of the resources. The horizontal scale controller is responsible for controlling the data transfers to rebalance the storage after adding/removing an instance. It calculates the amount of bandwidth allocated for the rebalancing operation as a tradeoff between a low bandwidth (which causes the operation to run unacceptably long) and a high bandwidth (which consumes application resources and increases its response time).

The proposal seems to be a feasible solution for storages which are able to rebal-

ance in a reasonably short time. If the storage is big and the operation takes a longer time, the system might not be able to react fast enough to an increase in the number of requests.

3 Summary

In this chapter we have presented several commercial cloud platforms. We have seen that Amazon EC2 [1] provides a mechanism for automatic scaling of an application. The granularity of such a scaling is set as a whole virtual machine. Amazon does not provide any automatic reconfiguration or data redistribution among the copies of the same machine. This has to be assured by the programmer.

Microsoft Windows Azure [5] can create several instances of one role and provides an interface to do it remotely. Unlike Amazon, it does not provide a controller to invoke the replication automatically. Nor does it provide any way to reconfigure the application and redistribute data.

Google App Engine [4] does not allow any replication of application components. The quota system is restrictive and does not allow for a high elasticity. The fixed quotas cannot be changed and the per-minute quotas prevent the application from flexibly reacting to short periods of high demand.

BOINC [7] and ASKALON [13] provide management systems for specific classes of applications. BOINC handles applications which can be divided into small workunits and processed independently to compose the final result. ASKALON supports applications which can be described as a set of components with interconnecting data flows. Both these systems use benchmarking to predict the behavior of application components and available resources.

The same holds for Demberel et al. [11], where an external manager assigns application components to resources based on the prediction of their needs.

Lim et al. [15] uses the concept of a Service Level Objective. Based on the current performance, the management system decides when a component needs to be replicated and handles application reconfiguration and data redistribution. The scope of the paper is however limited to storage applications.

Chapter III

Elastic Application Management

The aim of the project was to study the management of the lifecycle of an elastic application. In the first section of this chapter we state the problem. In the second section we give a detailed analysis. In the third section we explain the design decisions we made and the architecture of our solution. In the fourth section we describe how the solution was implemented. In the fifth section an example application is presented.

1 Problem Definition

In the context of our study, an application can be arbitrarily any software system, composed of several components. These components are defined at a certain level of abstraction – single objects, running program instances, virtual machines or even a set of physical machines.

The architecture of the application is known and can be precisely described. Application is composed of components and their interconnections. The architecture description defines in which order the components and interconnections should be started. Some of the components can be elastic, which means that several instances of the same component can be created and their number changed dynamically during the execution of the application. Interconnections with other components should be able to reconfigure themselves accordingly. Some of the components can be dynamic, which means that they can be started/stopped during the execution of the application. Elastic components are particular cases of dynamic components.

An *elastic manager* is a program which controls the whole lifecycle of the elastic application. It has to assure the execution of the following phases of the application:

- **Starting** – Create the components and interconnections in the order defined by the application architecture
- **Execution** – Observe the execution of the application and add or remove the instances of elastic components according to the actual load, start or stop dynamic components
- **Stopping** – Destroy the components and the interconnections

The elastic manager is composed of two parts:

- **Application-independent core** – Implements the functions which are generic

for all elastic applications
- **Application-specific code** – Implements parts of the management which are specific for the application, such as creation of the components, configuration and reconfiguration of specific types of interconnections, etc.

The aim of this research was to identify the requirements on such an application manager, evaluate different levels of abstraction at which it could operate and the consequences of this choice on the amount of code which could be implemented as application-independent and the code which would have to remain application-specific.

The expected output was a proposition of an architecture of such a manager and a demonstration of the feasibility of the concept with an example implementation.

2 Analysis

In this section we analyze several issues in the conception of an elastic manager and present different design choices which have to be taken into consideration. Namely we will present the choice of the component granularity, issues in making a component elastic, different ways of interconnecting the elastic components and the control of the elastic application.

2.1 Component Granularity

An important notion in the problem description is the concept of a *component*. Following the concept which is used in commercial cloud computing and which was presented in Section 1, the components could be defined at one of the following layers:

- **Application** – A component would correspond to a set of object instances in a programming language
- **Platform** – A component would be an instance of a running application
- **Infrastructure** – A component would be a whole virtual machine

Handling the elasticity at each of the levels of abstraction would have several advantages and disadvantages.

Application layer would provide the most detailed description of the application and allow to optimize the performance at the level of the smallest components. However, the complexity of such a description would easily become very high as it would require defining precisely which components should be elastic and the replication mechanism for each of the components and interconnection. Also such a solution would be restricted only to application running on one specific application platform.

Platform layer would operate with higher level description of the application – the model would become less complex and easier to define for a developer. However even in such an approach some requirements on the way application is implemented would remain. In order for a manager to be able operate and reconfigure the components, it would still have to follow the restrictions on the software platform used.

Infrastructure layer would give the most freedom for the developer. The application components would be the whole virtual machines running arbitrary software

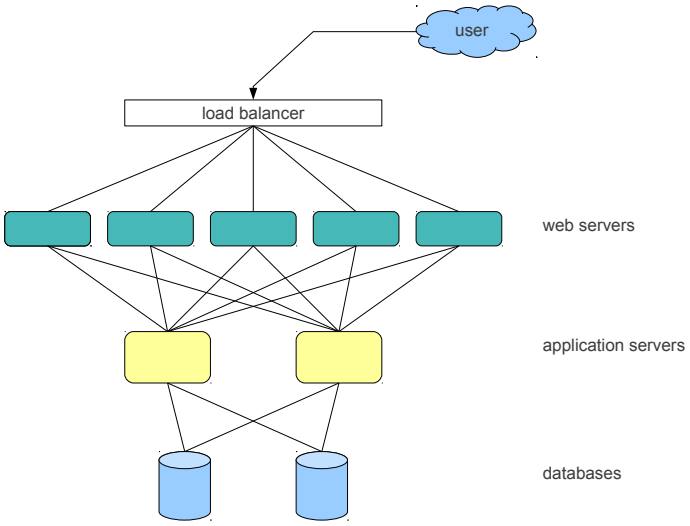

Figure III.1 – Three tier web application architecture

platforms. Such a solution on the other hand does not allow for very detailed description as the virtual machines are quite large execution units, which cannot be added or removed as fast as single application or even single object instances.

As we can see the advantages and disadvantages of the different choices follow the same pattern as in the cloud computing platforms. There is no single answer to the question which decision is the best as each of them might be suitable for different type of application.

2.2 Component Elasticity

With component elasticity we understand the problem of introducing a new instance of an existing component to a running application. If the application is to be elastic, it should be possible to insert or remove such an instance without affecting the rest of the application, it is without the need to stop all the other components.

A component of an application defined at any layer consists of two distinct parts – the execution code and the internal state. We suppose that the execution code for each of the component is available and does not change during the execution of the application. This is however not the case with the internal state. With respect to the internal state, two instances of the same component can be either independent or they might need to share at least part of their internal state.

Independent instances are easier to manage. It is sufficient to start a new instance without even notifying the existing ones. An example of such an instance is a web server, which only servers static content or generates dynamic pages (and does not support HTTP sessions) in a three tier web application architecture (see Figure III.1).

Dependent instances are much more difficult to manage. First it has to be determined which part of the internal state should be shared and which is instance-specific. In some cases it is easy to determine. For instance if the web servers in the above mentioned architecture had to support HTTP sessions, the only shared state informa-

tion would be the data associated with sessions. The other state variables associated with the generation of dynamic pages would remain with the instances. However, if we consider an arbitrary application component, such a distinction might not be obvious.

Once the shared state is determined, the sharing mechanism has to be implemented. There exists a variety of mechanisms from a shared memory or message passing interfaces to more complex ones. As can be seen in our example, in some cases a simple solution might work (a shared memory would be a feasible way of sharing session-related data between webservers), however in other cases, much more complex mechanism would have to be implemented (as in case of a replicated database in our example).

There is not one best way of implementing component elasticity and the choice of the mechanism will always depend on the specific application needs.

2.3 Interconnections

In order for the application to work, the application components need to exchange information. In our abstract concept, any communication channel between two components is called interconnection. Such an interconnection can be realized using an arbitrary mechanism, such as direct function call, shared memory, message passing interface, RPC, HTTP protocol etc.

Let us consider a successfully deployed application composed of components and their interconnections. When introducing new instances of an existing component, the already established connections have to be reconfigured in such a way that:

- the communication between existing components is not affected
- no data is lost in the application
- new instance of the component starts receiving messages in the same way as other instances of the component and gets integrated to the application in as short a time as possible

While these requirements can be fulfilled by a proper implementation of the reconfiguration mechanism, there is one issue which remains conceptual and requires cooperation of the application developer. It is the problem of the cardinality of interconnections. An interconnection might be of one of the following types:

1:n

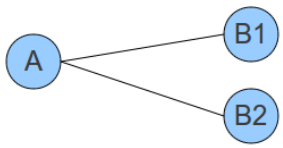

Figure III.2 – 1:n connection

This is a connection between one component and several instances of the other component (Figure III.2). In order to establish such a connection, the communication interface has to support such a configuration and a semantic of the communication

between the static component A and the elastic component B has to be defined. One possible semantic is that a message from component A will be delivered to both instances B1 and B2 (i.e. in case of updating configuration of two application servers). Another semantic is that the message is delivered only to one of the two instances (in a similar way as by a load balancer). This behavior might also be different for different messages. It is not possible to decide which mechanism to use without a detailed knowledge of the specific application.

m:n

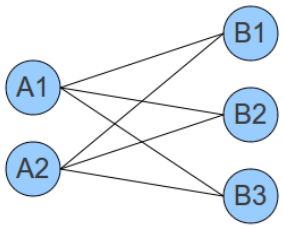

Figure III.3 – m:m connection

The establishment of communication channels between components in case of m:n connection (Figure III.3) is even more complex as it might not depend only on the knowledge of a sending component but on the state of the whole system. Figure III.4 shows some of the many configurations an interconnection between two components, each in 4 instances, can be established. Each of them might be valid in some case in some application and the decision again needs to be made by the application developer.

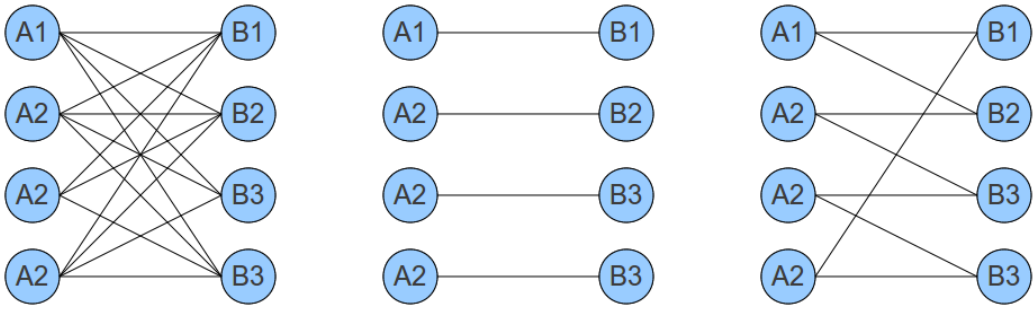

Figure III.4 – Different ways of establishment of communication channels in an m:n connection

2.4 Elasticity Control

As stated in Section 1 an elastic manager is a program which controls the execution of the elastic application. Such a manager operates from outside of the application – the manager knows the architecture of the application and can manipulate

its components and connections, while the application is not aware of the presence of the manager.

As we already stated, the elastic manager has three tasks – start the application, control its execution and stop the application. In this document we focus on the control part.

To control an elastic application means to:

– determine when to increase or decrease the number of instances of an elastic component
– realize the operation

As we have already presented in Chapter I, the main motivation for introducing elasticity is to improve the performance of an application by increasing the capacity of saturated components. The question of determining when a component is saturated and one more instance needs to be added, cannot be answered with one answer for all applications. The definition of saturation depends on a specific component and a specific application. As an example, it might be feasible to use a CPU usage as metric to evaluate saturation of a web server – as the number of requests per second increases, the CPU usage increases until at a certain point the server becomes saturated and a new instance should be added. In some other application, a limit on the number of threads executed in parallel can be imposed, etc.

The operation of creating or removing an instance of a component has to be implemented according to the layer of abstraction the components correspond to (see Section 2.1). There is a significant difference between the time and resources needed to create a new component at the different layers. Creating a new object in memory might only last few miliseconds, while starting a new virtual machine can take several minutes. This has also to be taken into account when implementing the elastic controller.

Again in this case we have identified some issues which are application specific and for which there is no universal solution.

3 Design

3.1 Architecture

In Section 2 we have analyzed the problem and identified the boundaries between the part of the system which can be used for an arbitrary elastic application and the part which needs to remain application-specific. We have seen that the application-specific part has to handle several relatively complex features, while the general part only handles the essential abstract notions of components and connections. We have also presented in the Introduction (Section 4) the notion of the self-configuration control loop.

These observations could be directly used to design an elastic manager. In order to make the usage of the manager more convenient, we have decided to use a modular design and support several additional features. Firstly, we wanted to make the management of elasticity independent on the process of starting the application. It is, to make it possible to start an application and join a controller of elasticity later if needed. Secondly, we wanted to separate the part responsible for adding and removing components from the decision-taking part to make it possible to implement several decision logics for the same application.

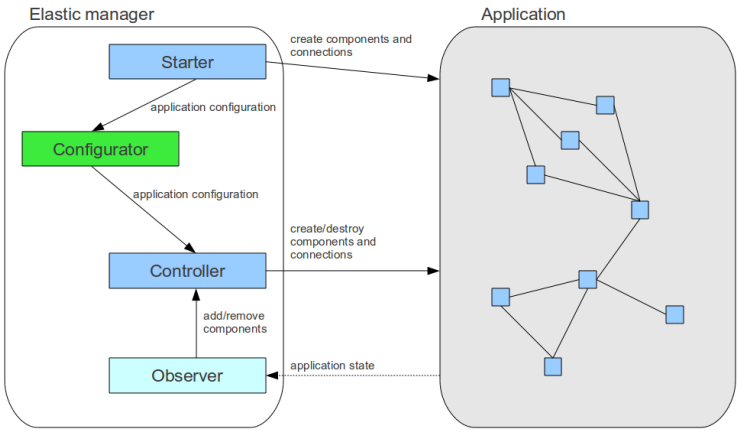

Figure III.5 – Components of the elastic manager

Our final design is composed of four interconnected servers (Figure III.5):

- **Starter** – Creates components and connections of the application
- **Configurator** – Stores the application configuration
- **Controller** – Creates dynamic components and changes the number of instances of elastic components
- **Observer** – Observes the runtime state of the application

In order to start the application, the configurator server has to be running. The starter server then reads the application description, creates the components and connections, transmits the information about the application architecture to the configurator and terminates.

To make the application elastic, first a controller server has to be started. It connects to the configurator and retrieves the application architecture. Second, an observer server connects to the controller, observers the runtime state of the application and asks the controller to adjust the components when needed.

The described architecture is an implementation of the self-control loop, where the sensor and decision parts of the loop are assured by the observer server, while the actuator part is assured by the controller. The system model (the current application configuration) is stored at the controller and retrieved by the observer. (Figure III.6)

3.2 Application Map

In order to start the application, the Starter component needs a description of the architecture. This description in our design is provided in the form of an application map – definition of components and their interconnections. In Section 1 we have already established three different kinds of components. In the design stage of the development of the elastic manager, we specify the components in more detail.

The description of a component provided by an application developer contains the following information:

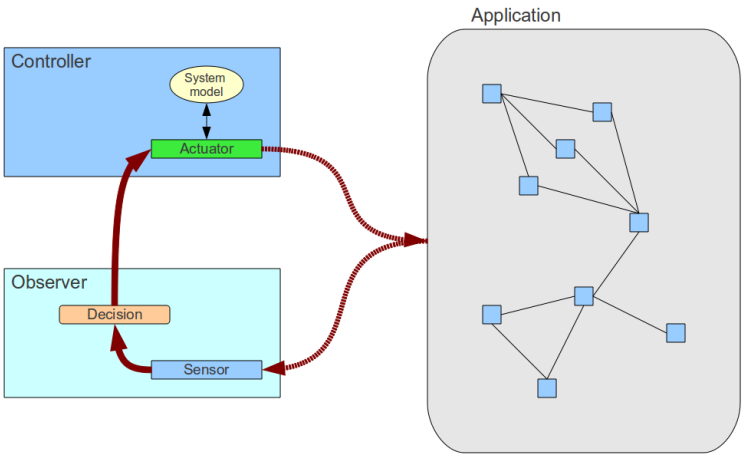

Figure III.6 – Self-configuration control loop within the elastic manager

- **Id** – a unique component id
- **Type** – type of the component, one of the following:
  - *Static* – the component can only be started at the beginning and destroyed at the end of the application lifetime
  - *Dynamic* – the component can be started/stopped during the execution of the application
  - *Elastic* – the component can exist in several instances (initial, minimal and maximal number can be set) and the number of instances can change during the execution of the application
- **Startup flag** – determines whether the component will be started on the application startup
- **Stage** – the number of the startup stage at which the component will be created

The description of a connection contains the following information:

- **Id** – a unique connection id
- **Type** – an application specific type of the connection
- **ComponentA** – id of the source component it interconnects
- **ComponentB** – id of the target component it interconnects

As we can see the application is described at a high level of abstraction. The application description can be extended with other information as required by the application-specific part of the implementation (see the next subsection), namely the type of the connection will always be defined by the needs of the application.

3.3 Application-specific Part

In order to make the elastic manager operational with a given application, the application developer has to provide an implementation of the following features:

- **Starter** and **Controller** servers – the following functions:
  - *startComponent(component)* – create instance(s) of a component as specified in the application description
  - *stopComponent(component)* – destroy instance(s) of a component
  - *reconfigureComponent(component)* – adjust the number of instances of an elastic component by creating new ones or removing existing ones
  - *bindConnection(connection)* – connect components with a connection as specified in the application description
  - *unbindConnection(connection)* – disconnect components by removing a connection
  - *reconfigureConnection(connection)* – adjust a connection which ends in an elastic component to the modified number of component instances
- **Observer** server – a control loop which will periodically check the status of the application (the sensor part), decide which action to take and send a request to the controller (the decision part)
- **Configurator** server – does not require any application-specific code.

It should be noted that the specification of the functions is given at a high level of abstraction. It does not specify any particular behavior with respect to different types of components and connections. It is up to the developer to define how the implementation will handle different choices in designing an elastic application presented in Section 2.

Particularly, the presented architecture does not impose any level of component granularity and it is a developer's decision whether the **startComponent** function will actually start a virtual machine, create an object instance or do something else.

The architecture does not specify any concrete way to handle elastic components and different cardinalities of interconnections between them. When a number of instances of an elastic component is requested to change, a **reconfigureComponent** function is first called. This function obtains a reference to an object describing the component – its type, configuration parameters and references to all existing instances. The **reconfigureComponent** function will create the new instance (or remove an existing one) and notify and reconfigure the other instances if needed.

Once this is done, the **reconfigureConnection** function is called for each connection going from or to the reconfigured component. The function obtains as an argument a reference to an object describing the component – its type, configuration parameters and source and target component description. Based on the needs of the specific application, the function will establish actual connections between components' instances and handle the 1:n and m:n connections. This function can also be used to execute on components functions that have to be executed after other components are created and the connection established.

4 Implementation

The elastic manager presented in this chapter was implemented in Java. The general part of the elastic manager constitutes package `org.ow2.ea`. It contains all the necessary classes needed to implement an elastic manager for a specific application.

The application developer is expected to inherit some classes from the `org.ow2.ea` package and to implement the application-specific parts. (See Figure III.8 for a class diagram of the `org.ow2.ea` package).

The four components of the elastic manager are distinct classes with their own `main` functions. They can therefore be executed as stand-alone applications on different machines. They use a protocol based on Java sockets for communication. The following functions can be invoked by other components on their interfaces:

- **Starter**
  - *start(String appMapFilename)* – Start the application described in the given file and send the map to the Configurator
- **Configurator**
  - *putMap(AppMap map)* – Store the application map
  - *getMap()* – Get the stored application map
- **Controller**
  - *getMap()* – Get the application map
  - *startComponent(String id)* – Start a dynamic component
  - *stopComponent(String id)* – Stop a dynamic component
  - *incComponent(String id)* – Increase the number of instances of an elastic component
  - *decComponent(String id)* – Decrease the number of instances of an elastic component

In the current implementation, the application map is read from a Java Properties file. Components and connections are identified by a unique String id.

5 Example

In this section we show how an elastic manager for a simple two-tier web architecture would be implemented. The application consists of a database and web servers. We suppose that the database is able to handle load much superior to any expected scenario and so there is no need to replicate the database. On the other hand, the web servers can get overloaded easily when the number of requests increases. It is possible to create several identical web servers and distribute the load among them using a load balancer. The architecture is depicted in Figure III.7.

We suppose that the application is running in a cloud and that:

- The components are stored in separate virtual machines
- It is possible to create several instances of the virtual machine with the web server
- It is possible to obtain a reference and connect to a newly created virtual machine
- There is no need to share state between web servers
- The load balancer is accessible from outside of the cloud and its address is known to users

In terms of our architecture, the application is composed of three components – two static components (**LoadBalancer** and **Database**) and one elastic component

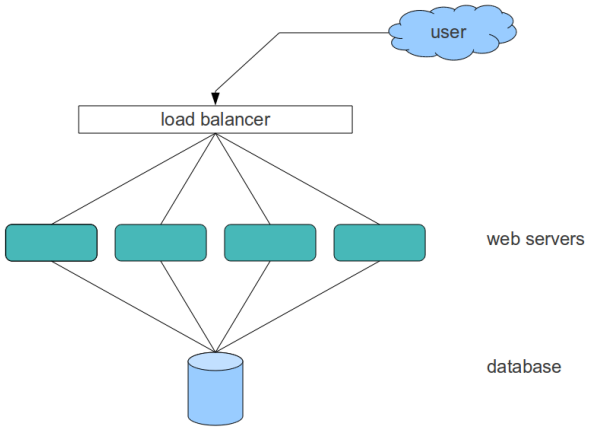

Figure III.7 – Two tier web application architecture

(**WebServer**). There are two connections – one between the database and the web server(s) (**DatabaseWebServer**) and one between the load balancer and the web server(s) (**LoadBalancerWebServer**).

The application map for such an application would look like the following (in form of a Java properties file):

```
appmap.nrComponents = 3
appmap.nrConnections = 2


appmap.component.0.id = LoadBalancer
appmap.component.0.type = static
appmap.component.0.vm.name = load-balancer
appmap.component.0.startup = true
appmap.component.0.stage = 3

appmap.component.1.id = WebServer
appmap.component.1.type = elastic
appmap.component.1.instances.min = 1      // the minimal number of instances
appmap.component.1.instances.max = 10     // the maximal number of instances
appmap.component.1.instances.def = 1      // the initial number of instances
appmap.component.1.vm.name = web-server
appmap.component.1.startup = true
appmap.component.1.stage = 1

appmap.component.2.id = Database
appmap.component.2.type = static
appmap.component.2.vm.name = database
appmap.component.2.startup = true
appmap.component.2.stage = 0
```

```
appmap.connection.0.id = DatabaseWebServer
appmap.connection.0.type = database-webserver
appmap.connection.0.componentA = Database
appmap.connection.0.componentB = WebServer
appmap.connection.0.stage = 2

appmap.connection.1.id = LoadBalancerWebServer
appmap.connection.1.type = load-balancer-web-server
appmap.connection.1.componentA = LoadBalancer
appmap.connection.1.componentB = WebServer
appmap.connection.1.stage = 4
```

As we can see, each of the components and connection is assigned a number of a stage at which it will be created. To start the application, the Starter will execute the following functions:

1. **startComponent(Database)** to start a virtual machine *database*
2. **startComponent(WebServer)** to start one virtual machine *web-server* (one is the default number of instances of the web server)
3. **bindConnection(DatabaseWebServer)** to wait until the database and web servers are started and connect the web server to the database
4. **startComponent(LoadBalancer)** to start a virtual machine *load-balancer*
5. **bindConnection(LoadBalancerWebServer)** to wait until the load balancer is started and connect the load balancer to the web server

After the last step, the application is ready and starts accepting incoming connections. The Starter server sends the application map to the Configurator server.

If desired, the Controller server for the application can be started. It will retrieve the application map from the Configurator and wait for incoming connections from an Observer.

The Observer, once started, will connect to the Controller, get the application map and connect to the existing web servers and load balancer. It will periodically check the load on the servers and if the load reaches a certain limit, it will call the **incComponent** function of the Controller.

To increase the number of instances of the WebServer component, the Controller will execute the following functions:

1. **reconfigureComponent(WebServer)** to create one more virtual machine *web-server*
2. **reconfigureConnection(DatabaseWebServer)** to wait until the new virtual machine is created and connect the new web server to the database
3. **reconfigureConnection(LoadBalancerWebServer)** to connect the new web server to the load balancer and reconfigure the load balancer

The processes of decreasing the number of instances and stopping the application are reverses of the described processes.

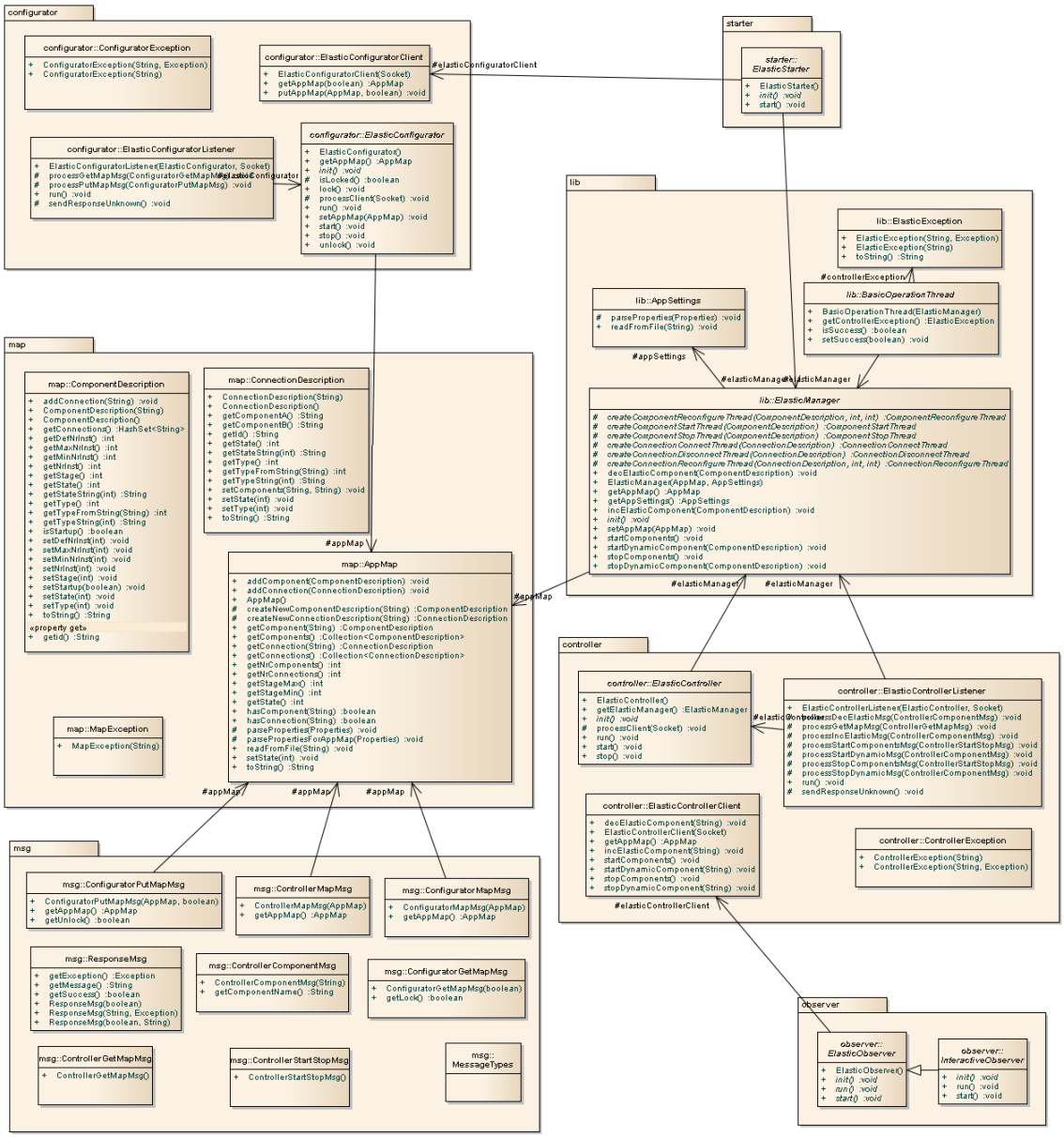

Figure III.8 – Class diagram of the org.ow2.ea package – the part of the elastic manager which is common for all applications

Chapter IV

Use Case

This chapter presents the implementation of an elastic application manager for a load injection application Clif [2]. The first section explains why this use case was selected. The second section introduces the problem of load testing. The third section presents the architecture of the Clif application and the fourth section describes the elastic application manager for this application.

1 Motivation

In the previous chapter we have showed that in case of some application and some design choices the problem of making an application elastic can become very complex. We were therefore looking for an application which would make the process less complicated, while still allow us to show the main features and the feasibility of the approach.

In cooperation with our industrial partner Orange Labs we decided to use a load injection application Clif as a use case. Clif has several advantages for our purposes. It is written in a component-oriented framework, thus the identification and separation of components is already done. Furthermore, there is no complex communication between components which need to be elastic. And finally, the performance of such an application can be easily measured for further analysis.

2 Load Testing

The idea of load testing is to check that the behavior of a given system under test (SUT) conforms to specification. A common approach is to send requests to the SUT, wait for replies and measure response times as they are experienced by the user or a client system.

Traffic generators are often referred to as **load injectors**. Several load injectors might be used at the same time to generate heavy workload. The generated workload may emulate the traffic of a number of real users of clients through so-called **virtual users**.

Probes are used to obtain accurate measurements of performance at the injectors and the SUT. These measurements include characteristics such as CPU or memory consumption and can be used for tuning both the SUT and the injectors.

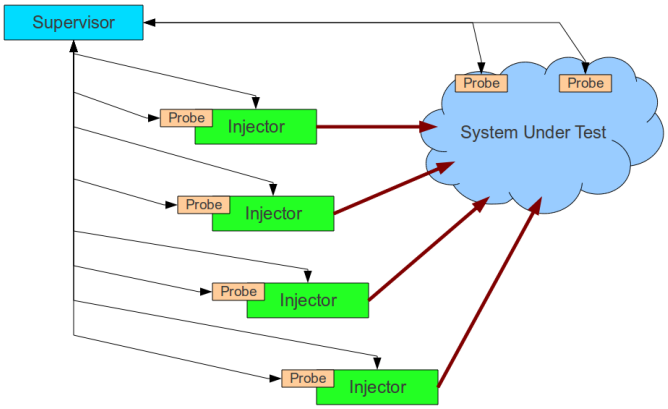

Figure IV.1 – Scheme of a load testing platform

A **supervisor** is in charge of controlling and monitoring the distributed set of load injectors and probes. Figure IV.1 shows a scheme of a load testing platform.

3 Clif

3.1 Presentation

Clif is a load injection framework which has been jointly developed by France Telecom and INRIA in the context of the Java Middleware Open Benchmarking Initiative – an initiative dedicated to benchmarking and performance issues in the context of the ObjectWeb open source community. The idea was to provide a generic, scalable and user-friendly platform for load injection and performance reporting [12].

Clif platform provides several types of load injectors for generating traffic supporting common protocols such as HTTP, FTP, SIP, etc. For measuring resource usage several probes are implemented for processor and memory consumption, network traffic, etc. Clif provides tools for supervising running tests and analysis tools. It can be controlled via a command line interface or one of several graphical interfaces (including an Eclipse plug-in).

Clif is an open source application and can be extended by implementing new injectors and probes in Java language.

3.2 Component Model

Clif is based on Fractal. Fractal is a component model, which was also developed by the ObjectWeb community [3]. One of the motivations for this decision was to evaluate Fractal's support for distributed applications and its claimed flexibility [12].

Main goals of the Fractal component model were to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. The main features of Fractal are:

- Composite components – to provide a uniform view of applications at various abstraction levels
- Shared components – to model resources

– Introspection capabilities – to monitor a running system
– Configuration and reconfiguration capabilities – to deploy and dynamically reconfigure an application

Another goal was to be applicable to many software, from embedded software to application servers and information systems [9].

Fractal component model is defined as an extensible system of relations between well defined concepts and corresponding APIs that Fractal components may or many not implement. This set of specifications is organized as increasing *levels of control*.

At the lowest level, a Fractal component is a runtime entity that does not provide any control capability to other components and is therefore like an object.

At the next level, a component provides a standard interface that other components can use to discover all its external interfaces (external introspection).

At the last level, the component allows other components not only to discover, but also to modify its *content*, i.e. what is inside the component. In the Fractal model, this content is made of other Fractal components, called its *subcomponents*, bound together through *bindings*. [10]

Component discovery is assured by using *Fractal registry*. It is a standalone application running at a network location, whose address is known to all components of the application. When a Fractal component is created, it can register itself in the registry using a unique string id. Other components can later use the id to retrieve a reference to the component.

With Fractal it is possible to deploy components at another physical machine. Two special Fractal servers have to be running at known locations – the *registry* and the *code server*. The process works as follows: First, the target machine creates an empty *Fractal server* and registers it in the registry. The source machine connects to the registry and retrieves reference to the server. Then it requests the server on the target machine to deploy the component. The target Fractal server connects to the code server, downloads component bytecode and resource files, creates the component and registers it in the registry.

3.3 Architecture

The architecture of Clif is based on the architecture presented in Section 2. It relies on five component types (Figure IV.2):

– Load injector components
– Probe components
– One supervisor component
– One storage component to store all measures and test plan definitions
– One or several analysis components that get measures from the storage component and provide performance analysis and reporting facilities

Load injectors and probes are autonomous in their activity and are controlled by the supervisor. They produce data (measures, lifecycle events, alarms,...) that are retrieved by the storage component at the end of the test component. Monitoring is achieved in the following way: each load injector and probe maintains moving statistical values about their activity. A supervisor can request the values when needed using the component's **DataCollector** interface.

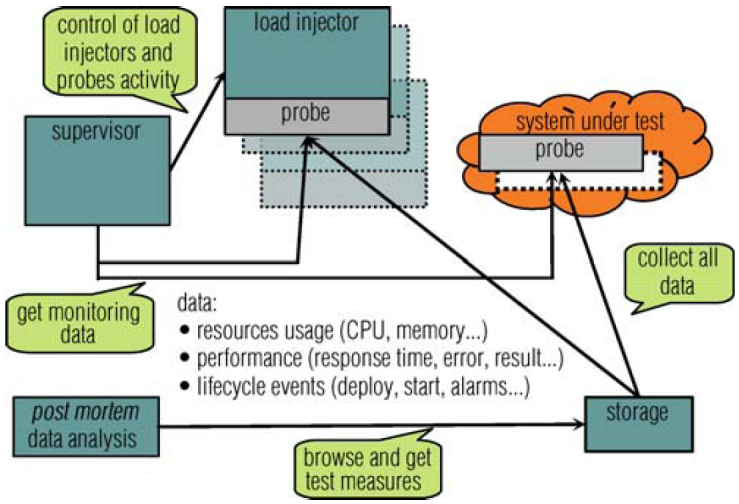

Figure IV.2 – Clif's top level architecture [12]

In Clif, due to their similarity, the load injectors and probes are considered a unique component type – the *blade* type. Blades contain arbitrary computation capabilities that can be controlled and monitored by the supervisor component. Furthermore they conform to a well-specified lifecycle. A blade component is first deployed, then initialized and started, and then possibly suspended and resumed. At the end of its activity, the blade can be either aborted (in case of error), completed (in case of a successful execution) or stopped (from outside).

In the context of load testing, the deployed blades are first initialized so that they become ready to start as soon as they are requested to. This leaves enough time to prefetch libraries (i.e. Java classes), get data sets from files etc. When all the blades are ready, the supervisor requests them all to start their activity. The activity may or may not terminate by itself, either because the workload scenario (respectively the scheduled observation duration) has completed or failed, or it may be terminated by a stop request. The supervisor then asks the storage component to collect the results from the blades.

4 Elastic Clif

4.1 Specification

In our use case we focused on measuring the maximal throughput of a web application, i.e. the task was to find out the maximal number of requests a web application is able to handle before it becomes saturated and its response times deteriorate significantly. To do so, it is necessary to use a load injector which generates traffic to the application – the *system under test* (SUT). The traffic injector consumes significant amounts of physical resources of the machine it is running on and it is often needed to use several injectors running on different machines. It is usually difficult to determine the result in advance and set the number of injectors accordingly. A possible solution to this problem might be to use an elastic approach – to gradually increase

the number of injectors until the server is saturated.

The details of performance evaluation of web applications were out of scope of our project and we did not aim to find an exact method to do such an evaluation. Instead, we focused on the challenge of making the Clif application elastic. The aim of our work was therefore to develop an elastic manager for Clif which would be able to adjust the number of injectors during the execution of a test upon a request from an elastic observer.

4.2 Application Architecture

In order to make an application elastic, one has to find out which components the application is composed of, what are their interconnections and which components should be made elastic. With Clif, this is easy to determine as the application itself is already component-based and the Fractal component model allows to distribute the components to several locations and handles the communication.

As we have seen in Section 3.3, Clif application is composed of three main components: the *supervisor*, the *data storage* and the *load injector*. Out of these components, only load injector has to be elastic.

The supervisor component is accompanied with data files needed to execute a particular test – namely a *.cls* file describing the probes and injectors and a file defining the HTTP injection scenario.

In addition to these components, two servers of the Fractal component model have to be running – a registry, which is used for the components to be able to discover other components and a code server, which is used to distribute the bytecode of the components to different machines.

Another important design decision is about the platform and component granularity. In case of Clif one could envisage a platform as a service (PaaS) platform which would provide support for running single Fractal components. However, as no such a platform exists at this time, we have decided for an infrastructure as a service (IaaS) approach and distribute Clif components to several standalone virtual machines.

We did not find a particular reason for separating the supervisor and the data storage to different machines. With respect to elasticity, our application is therefore composed of two components:

- **clif-supervisor** (static) – contains the *supervisor* and *data storage* components, the Fractal registry and code server
- **clif-injector** (elastic) – contains the *load injector*

For the needs of the application it is sufficient to define just one interconnection between the components – **clif-supervisor-injector**. The default number of instances of the elastic component is set to one.

4.3 Execution Environment

We used a set of physical machines connected to a local network to simulate an IaaS cloud platform. To create such a platform, two essential components are needed – a hypervisor to launch virtual machines and a convenient interface for applications to control the instances of the virtual machines.

4.3.1 Hypervisor

Nowadays, there are several hypervisors available. As the particular choice of the hypervisor did not have effect on our experiments, we did not need to do any further analysis and used Xen, which had been already used in the SARDES team. **Xen** is a hypervisor for IA-32, x86-64, Itanium and ARM architecture. According to its authors, it is an open source industry standard for virtualization [6]. Xen systems have a structure with the Xen hypervisor running as the lowest and most privileged layer. Above this layer run one of more guest operating systems, which the hypervisor schedules across the physical CPUs. The first guest operating system, called *dom0* boots automatically and has special management privileges. The system administrator can log into dom0 to manage any other guest operating systems.

Xen virtual machines are stored on the hard drive of the physical machine and consist of several files – a configuration file which contains parameters of the machine (such as its name, network address, etc.) and the location of other files, which contain the images of the virtual hard drives of the machine. A user can start a machine by simply entering the following command on the console of the *dom0* system:

```
xm create <configuration_file>
```

Once the machine is started, the user can connect to the console of the machine and control it in an interactive way. In the cloud computing context, the machines are expected to start and run autonomously without the need for any action from the administrator. They would therefore contain all the required initialization in their startup scripts.

A pure Xen-based platform has two limitations for the needs of our simulated IaaS. Firstly, it is not possible to start several instances of the same machine (i.e. to use the same configuration file and disk image to create two instances) and secondly, it is not possible to pass arguments to the machine upon startup.

4.3.2 Cloud Interface

To overcome these limitation, we have decided to develop a Simple Cloud Interface (**Sclint**). Sclint is a server application running on the *dom0* system. It accepts connections on a predefined port and uses a Java-socket based protocol to communicate with clients.

When started, the Sclint server reads a configuration file with a list of *instance types* and *instances* which are available at the machine. An *instance type* is an entity defined by its unique String id and consists of several *instances*. An instance corresponds to one Xen virtual machine.

A client can request Sclint to start a new instance. It does so by sending a *Create* command with two parameters – the *instance type* id and the arguments. Sclint server then starts one of the instances of the requested instance type which is not yet running. When the instance is started, it can connect to the Sclint server and obtain the arguments it was started with.

The Sclint implementation also provides a simple interactive console based client.

4.3.3 Elastic Manager

The elastic manager servers are running on a separate machine(s). The elastic starter can be located outside the cloud and only needs to be able to connect to the Sclint interface and to the running elastic configurator.

The elastic controller and elastic observer have to be located in the cloud as they need to have direct connection not only to the Sclint interface, but also to the running application components.

4.4 Implementation

Chapter III presented the general architecture of an elastic application manager and gave several guidelines on implementing an elastic manager for a specific application. It was stated that the application developer is supposed to inherit several classes from the `org.ow2.ea` package and implement several functions. Furthermore the startup behavior of the components (virtual machines) has to be defined.

This example also shows that the notion of interconnection in our systems does not necessarily correspond to a physical interconnection. Instead it can be used to carry on operation that can only be done once the two components are fully initialized or to move part of the operation to an external entity.

4.4.1 Machine Startup

Defining the startup behavior of the machines is straightforward from the description in Section 4.2. Each of the machines has to initialize the components it contains.

clif-supervisor Upon startup, the supervisor machine starts the two servers needed to run a Fractal application (registry and code server). For the sake of simplicity, we assigned a fixed IP address to the machine to make sure that the servers will always be accessible at the same address. When the registry is ready, the *supervisor* and *data collector* components are created. The test plan (.cls) file is loaded to the memory and the components registered to the registry. From now, the supervisor machine is waiting until all the probes from the test plan are deployed on servers by an external entity. Once it happens, it starts the execution of the test.

clif-injector-server This machine is supposed to contain the *clif-injector*. However in Clif, the injectors are defined in the test plan (.cls) file and deployed once the test is started. During the machine startup this file is not available. Instead, the startup script will create an empty Fractal server and connect to the Sclint interface to obtain the startup arguments. It will register the server in the registry under the name obtained as a value of the `clif-name` argument.

4.4.2 Elastic Starter

The elastic starter works in three steps – it creates the *clif-supervisor* machine, the *clif-injector-server* machine and the *clif-supervisor-injector-server* interconnection.

1. To create the *clif-supervisor* component, the starter will request the *Sclint* interface to start one instance of type *clif-supervisor*. It will then periodically check the address of the Fractal registry until the registry appears and the *supervisor* component is registered.

2. The starter will request the *Sclint* interface to create one instance of type *clif-injector-server* and pass an argument `clif-name` as defined in the application map. When the new Fractal server appears in the registry, the machine is considered started.

3. The starter will obtain from the registry a reference to the *supervisor* component and get the test plan. It will read the test plan and deploy the blades on the Fractal servers accordingly. Then it will bind the new components to the *supervisor* and *data collector* components.

It should be noted that after step 3. the control of the application execution goes back to the *supervisor* component, which starts the test execution.

At this moment, the test is running according to the test plan. The starter sends the application map to the configurator and terminates.

4.4.3 Elastic Controller

The only operation supported by the elastic controller is increasing the number of instances of the *clif-injector-server* component upon request from an observer. This operation consists of two steps – creating new component and reconfiguration of the connection with supervisor.

1. As explained in Section 4.3.2 the Sclint interface supports creating of several instances of the same instance type. The only problem in case of our application is that the registry requires servers to be registered with a unique name. The elastic controller uses the following convention for assigning the value of the `clif-name` argument: For the first instance, the value is passed as defined in the application map. For the second instance, `-2` is appended at the end of the name, for the third instance `-3` etc. As before, the component is considered started when the new server appears in the registry.

2. The controller connects to the *supervisor* and obtains the test plan. It check for the blades that had been deployed on the first instance. For each of the blades a new instance has to be created in deployed on the new server. As the blade names also have to unique, the controller uses the same convention as for the servers to name the blades. It deploys the blades, binds them to the *supervisor* and *data collector* and checks for the state of the already existing blades. It changes the state of the new blades to match the state of the existing ones (i.e. if the existing blades are in state RUNNING, it will initialize and start the new blades too).

The operation of adding new blades does not take into account the internal state of the existing ones and that existing blades are not notified of the existence of the

new ones. This is a design decision motivated by the fact that in case of Clif no such synchronization is needed.

4.5 Elastic Observer

The task of the elastic observer is to monitor the execution of the test and adjust the number of instances of elastic components accordingly. As the performance testing was not in scope of our project, we did not implement an automatic control loop for a particular test. We implemented a simple scenario, where the number of elastic instances is increased in predefined time intervals.

To collect intermediate data from the probes, the observer is running a separate thread, which periodically connects to the *supervisor* and obtains the list of blades. It then connects to the `DataCollector` interface of the blades and obtains the values available. This data could be used by an automatic control loop to take decision. In our implementation, they are stored in .csv files for further analysis.

4.6 Stopping the Application

The observer is responsible for issuing a command to stop the application. It sends a command to the controller, which has to stop the connection and then the components.

In the connection stopping phase, the controller will connect to the *supervisor*, send a `stop` and `collect` commands for the supervisor to collect the data from the probes and save them on disk on the supervisor machine.

To stop the components, the controller will call Sclint to destroy the running instances.

The data obtained by the `collect` operation can be found in files on the supervisor machine. The data obtained by the observer are located in files on the machine running the observer.

Chapter V

Evaluation

In this chapter we present results obtained when measuring the maximum throughput of a web application. In the first section we explain the test configuration, in the second section the obtained results are presented and in the third section we summarize our observations.

1 Configuration

As an example web application for our experiment we used MyStore. MyStore is an application developed at France Telecom for experiments with performance testing. It simulates behavior of an usual online shop (listing items, adding to cart etc.). MyStore is written in Java and runs on application server JOnAS. It is distributed as a sample program with a deployment tool JaDOrT. All these applications are parts of an open source stack for enterprise computing developed by the ObjectWeb consortium (nowadays OW2). The system was installed in a virtual machine in the the cloud.

We used a configuration with maximum of five injectors. Each injector generates traffic corresponding to 20 virtual users per second. As we already stated in Chapter IV we did not implement an automatic control loop. Instead we started with one injector and were increasing the number of injectors five minutes after the previously inserted injector was initialized.

Apart from the injectors, the measurements were also obtained from two probes (CPU and RAM) deployed directly on the machine with the web application.

We used two physical machines for our experiments. The injectors were running in virtual machines on physical machine `scloud02`, the web server in a virtual machine on physical machine `scloud01`. The physical machines were dual core processor machines with Intel Core Duo 1.66 GHz, 2 GB RAM and Linux 2.6.32 and Xen 3.0 installed. The virtual machines were each assigned 256 MB RAM and were running Linux 2.6.18 and Java Virtual Machine version 1.6.0. The physical machines were connected via a 10 Gb switch in a separate network isolated from other network traffic.

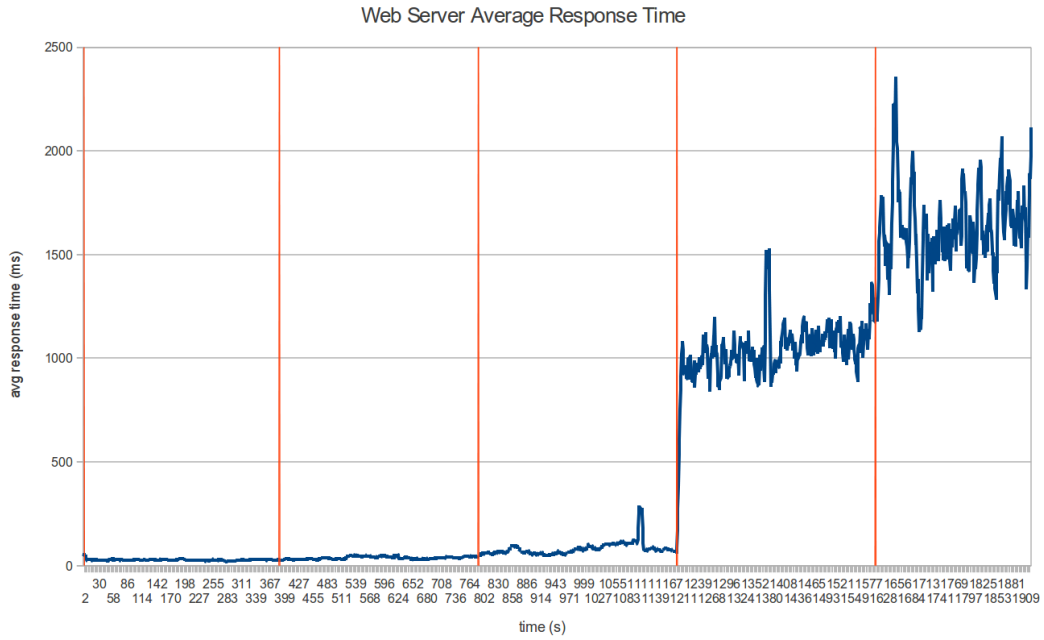

Figure V.1 – Average response time of the server. Vertical lines show times when a new injector was inserted.

2 Results

The important measurement was the response time of the server. Figure V.1 shows the values obtained. It can be seen that the response time remains relatively low as long as maximum of three injectors are present. After inserting the fourth injector, the response time increases dramatically to a level which is not acceptable for most users (around 1000 ms instead of 30 ms at the beginning).

Figure V.2 explains what happens in the server. When one injector is present, the CPU usage is constant at the level of around 19 %. After inserting the second injector, it doubles and starts slightly linearly increasing. When the third injector is added, the usage again increases significantly and the linear increase becomes faster. This is caused by the resource contention when the number of users is becoming too high. After inserting the fourth injector the CPU usage reaches 100 %.

Figure V.3 shows similar increase in the usage of RAM. In this case, after inserting the fourth injector, the usage remains at 93 %.

The results show that the system under test starts having serious problems when more than 60 virtual users per second are present. When the number of virtual users reaches 80, the system is saturated and the time of response is not acceptable anymore.

3 Observations

The experiment shows that elasticity on injectors is a feasible approach to performance evaluation. The tested system was running on a not very powerful machine, so

the number of injectors needed was low. However the approach is not limited in size and could be used on a much larger infrastructure with several hundreds of injectors.

A drawback of the current implementation is the time it takes to insert new injector. When the controller is asked to insert new instance of the injector component, it has to start a new virtual machine, wait until it is started and initialized. With our infrastructure this operation took on average 85 seconds.

To make this time shorten, it would be possible to modify the mechanism so that the controller would keep several machines started in advance and only initialize and start the blades when requested. This way the time of reaction could be lowered to few seconds.

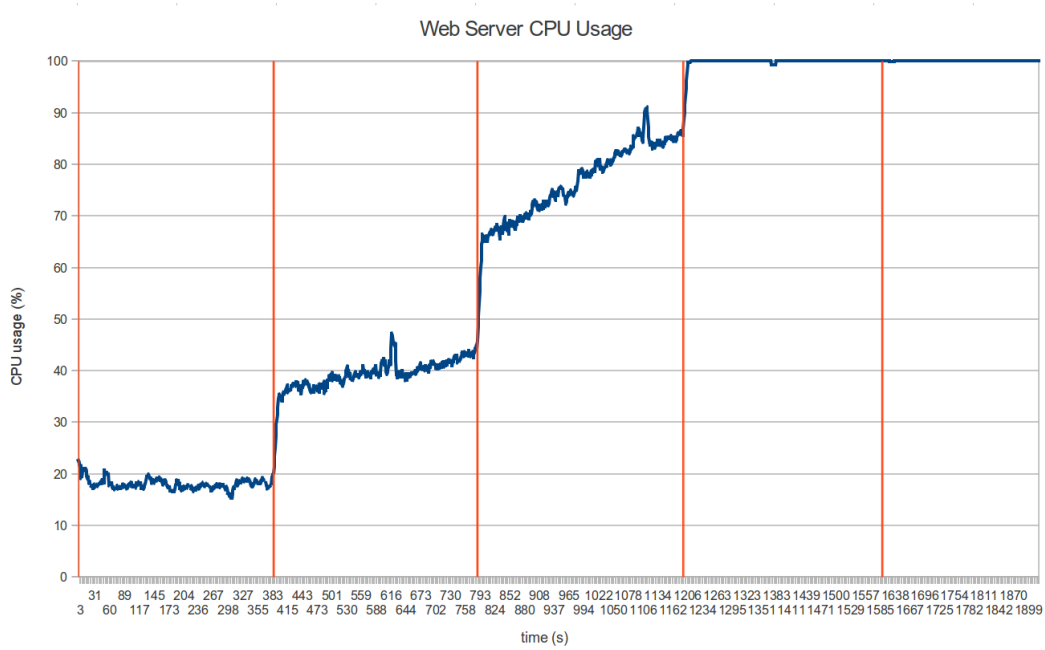

Figure V.2 – CPU usage of the server. Vertical lines show times when a new injector was inserted.

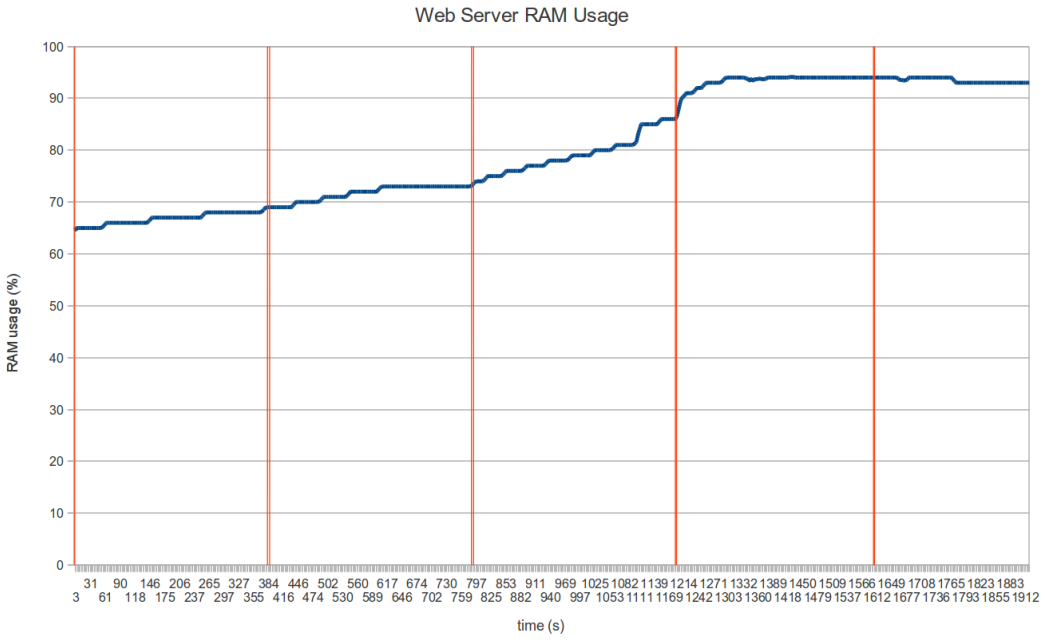

Figure V.3 – RAM usage of the server. Vertical lines show times when a new injector was inserted.

Chapter VI

Conclusion

Cloud computing has been becoming increasingly popular within business world in recent years. Several industrial providers run large scale cloud computing platforms offering different services at various levels of infrastructure, platform and application. One of the problems that is not yet fully resolved is the problem of making applications elastic, it is to make them automatically adjust to variations in load without the need of intervention of a human administrator and without the need to change the code of existing applications.

In the context of autonomic computing, which is an effort to develop self-managed complex software systems, the problem can be stated as a self-optimization problem. Autonomic computing suggests a solution using autonomic managers – an external application, which continuously observes performance of the application and reconfigures it to better handle actual needs.

In this project, we first analyzed the challenges faced when implementing elastic control of an arbitrary application. We have identified several design choices that have to be made when making an application elastic: the choice of granularity of components, determining which components are elastic, handling interconnections of elastic components and the actual control of elasticity. We then presented an architecture of an elastic manager with a well defined separation between general concepts and application-specific parts.

To show the feasibility of the approach, we have selected a load injection application Clif. We studied the Fractal component model used for development of the application, analyzed its architecture and identified elastic components. In an experiment with a web application we have shown that the approach is working and can be used for development of elastic controllers for real-world applications.

There are two possible directions of future work on the topic: One would be to continue with the Clif experiment and implement and evaluate a fully automatic control loop which would be able to control a larger scale of experiments. The other direction would be to move the control from the infrastructure layer to the platform layer and develop an elastic manager which would work directly with Fractal components.

Bibliography

[1] Amazon EC2, May 2011. URL `http://aws.amazon.com/ec2/`.

[2] Clif, May 2011. URL `http://clif.ow2.org/`.

[3] Fractal, May 2011. URL `http://fractal.ow2.org/`.

[4] Google App Engine, May 2011. URL `http://code.google.com/appengine/`.

[5] Microsoft Windows Azure, May 2011. URL `http://www.microsoft.com/windowsazure/`.

[6] Xen, May 2011. URL `http://www.xen.org/`.

[7] D.P. Anderson. BOINC: a system for Public-Resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, PA, USA.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. 2009.

[9] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, September 2006.

[10] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model. February 2004. URL `http://fractal.ow2.org/specification/index.html`.

[11] Azbayar Demberel, Jeff Chase, and Shivnath Babu. Reflective control for an elastic cloud application: an automated experiment workbench. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 8–8, San Diego, California, 2009. USENIX Association.

[12] Bruno Dillenseger. CLIF, a framework based on fractal for flexible, distributed load testing. *Annals of Telecommunications*, 64(1-2):101–120, 2008.

[13] T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh Truong, A. Villazon, and M. Wieczorek. ASKALON: a grid application development and computing environment. In *Grid Computing, IEEE/ACM International Workshop on*, pages 122–131, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[14] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceeding of the 7th international conference on Autonomic computing - ICAC '10*, page 1, Washington, DC, USA, 2010.