DESIGN AND MANAGEMENT OF DISTRIBUTED SELF-ADAPTIVE SYSTEMS

Doctoral Dissertation of:
Luca Florio

Supervisor:
Prof. Elisabetta Di Nitto

Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2016 – XXIX
I would like to thank my family, that supported me in every choice I made during my life.

Thank you to my girlfriend Eleonora, that has been my light in the darkest moments.

Thank you to my friends, that have always been there.

Thank you to my Advisor Elisabetta, that guided me in this journey and helped me to achieve all the results I obtained in my research.

Thank you to Danilo and Matteo, that helped me in my teaching experience and that always gave me good advices.

Thank you to Michele and Roberto, with whom I shared the office, and that have always been super-nice.

Thank you to all my colleagues, that made every day I spent at Politecnico a very pleasant day.

*Thank you to my willpower, that always makes me achieve my goals.*

Luca Florio
Abstract

SELF-Adaptation is the capability of a system to adapt in an autonomous way to every change in the scenario in which it operates. This capability is fundamental in distributed systems, since they are composed of thousands of elements that work in a very dynamic and unpredictable environment. Distributed systems play a central role in the current software development landscape: the advent of technologies like cloud computing changed the development methodologies, moving from applications built as a single and monolithic piece of software to systems composed of a great number of decoupled elements distributed on a large scale. Thus, distributed Self-Adaptive Systems are an actual need and require to be studied in order to understand the most effective way to design and manage them.

The literature provides different approaches to deal with such systems. However, while several algorithms for Self-Adaptation have been proposed, stable platforms and comprehensive software engineering approaches for the application of such algorithms to a concrete context are still to come. This thesis addresses these challenges, providing methods and tools to design and manage distributed systems able to autonomously adapt to changes and operate in a concrete and dynamic context.

We started from the engineering of an existing decentralized self-adaptive system called the SELFLET Framework, that has been improved and deployed in a Cloud Computing environment in order to be evaluated not through simulation, but with a concrete case study. Despite the results of the evaluation highlight the capability of the SELFLET Framework to adapt a cloud application to the changing workload it can face, we identified some
issues for its concrete adoption by practitioners: the use of a framework re-
quires to learn how to use it, and imposes constraints on the technologies
adopted for the development of the application. This led to the study and
design of a distributed Self-Adaptive System that inherits all the strengths
of the SELFLET Framework (decentralization, emergent behavior, encaps-
sulation of the business logic) but that is as decoupled as possible from the
technology stack of the application that should be managed.

In order to validate our approach we designed and implemented GRU,
a tool that exploits the concept of virtual container to bring self-adaptive
capabilities to applications developed using the Microservices Architecture
pattern and deployed in Docker containers: these technologies have been
recently adopted by the major companies in the IT industry and represents
a good fit for our prototype. We evaluated the prototype of GRU with a con-
crete case study, deploying the system in a cloud infrastructure composed
of several nodes. Despite the limitations of our prototype, the results of
the evaluation show that GRU can adapt the application to the changes in
the workload, and validate our approach for the transparent adaptation of a
distributed system.
Sommario

SELF-Adaptation è la capacità di un sistema di adattarsi in maniera autonoma ai cambiamenti nello scenario dove opera. Questa capacità è fondamentale per i sistemi distribuiti, essendo composti da migliaia di elementi che operano in un ambiente dinamico ed imprevedibile. I sistemi distribuiti giocano un ruolo centrale nell’attuale panorama dello sviluppo software: l’avvento di tecnologie quali il cloud computing hanno cambiato le metodologie di sviluppo, passando da applicazioni sviluppate come un unico blocco a sistemi composti da un gran numero di elementi distribuiti su larga scala. I sistemi Self-Adaptive distribuiti sono dunque una necessità reale e richiedono di essere studiati per comprendere il modo più efficace per progettarli e gestirli.

In letteratura è possibile trovare diversi approcci per la realizzazione di tali sistemi. Tuttavia, mentre sono stati proposti svariati algoritmi per la Self-Adaptation, mancano tuttora piattaforme ed approcci di ingegneria del software per una loro efficace applicazione a contesti concreti. Questa tesi affronta queste sfide, fornendo metodi e strumenti per progettare e gestire sistemi distribuiti in grado di adattarsi autonomamente ai cambiamenti imprevisti e operare in un contesto reale e dinamico.

Il nostro lavoro è partito da un sistema Self-Adaptive decentralizzato esistente chiamato SELFLET Framework, che è stato migliorato e testato in un’infrastruttura di cloud computing al fine di essere valutato non attraverso una simulazione, ma con un caso di studio concreto. Nonostante i risultati della valutazione evidenzino le capacità del SELFLET Framework, abbiamo identificato alcuni vincoli che potrebbero limitarne l’adozione da parte

Al fine di validare il nostro approccio abbiamo progettato e implementato GRU, un tool che sfrutta il concetto di virtual container per integrare capacità di Self-Adaptation in sistemi sviluppati utilizzando l’architettura a microservizi e distribuiti in Docker containers: queste tecnologie sono state recentemente adottate dalle principali aziende del settore IT e rappresentano un buon banco di prova per il nostro prototipo. Abbiamo valutato il prototipo di GRU con un caso di studio concreto, eseguendo il sistema in un’infrastruttura cloud composta da diversi nodi. Nonostante le limitazioni del nostro prototipo, i risultati della valutazione sono promettenti e validano il nostro approccio per l’integrazione trasparente della Self-Adaptation in un sistema distribuito.
Contents

1 Introduction 1
  1.1 Self-Adaptive Systems .......................... 2
  1.2 Self-Adaptation in Distributed Systems ............ 3
  1.3 Research goals .................................. 5
  1.4 Contributions ................................... 6
  1.5 Structure of the thesis ............................ 10

2 State Of The Art 13
  2.1 Self-Adaptation .................................. 13
    2.1.1 Self-Adaptation properties .................. 14
    2.1.2 Self-Adaptation implementation .............. 17
  2.2 Cloud Computing ................................ 22
    2.2.1 Cloud Applications Self-Adaptation .......... 23
    2.2.2 Resource Management Self-Adaptation ........ 25
  2.3 Conclusion ...................................... 25

3 The SELFLET Framework 27
  3.1 Overview ........................................ 29
  3.2 Architecture .................................... 31
    3.2.1 Services .................................... 31
    3.2.2 Behaviors ................................... 32
    3.2.3 Autonomic policies ......................... 33
  3.3 Adaptation Loop .................................. 36
    3.3.1 Action Generation ............................ 36
Contents

3.3.2 Action Selection .................................. 40
3.4 Internal architecture .................................. 41
3.5 Conclusion ........................................... 43

4 Evaluation of the SELFLET Framework ................. 45
4.1 Case study ............................................ 46
4.2 Evaluation ............................................ 50
  4.2.1 Setup ........................................... 51
  4.2.2 Experimental results .............................. 55
4.3 Discussion ............................................ 59
4.4 Conclusion ............................................ 61

5 Enabling Transparent Adaptation in Complex Systems ... 63
  5.1 Advantages and limitations of the SELFLET Framework ... 64
  5.2 A tool to enable transparent adaptation in complex systems ... 68
  5.3 Technological considerations ......................... 69
    5.3.1 Adaptation manager ............................ 70
    5.3.2 Managed element: Microservices Architecture .... 70
    5.3.3 Adaptation enabler: Docker containers ............ 72
  5.4 Self-Adaptation in Microservices Architecture and Docker containers .......................... 72
    5.4.1 Microservices Architecture Self-Adaptation .... 73
    5.4.2 Docker containers Self-Adaptation ............... 74
  5.5 Conclusion ............................................ 76

6 GRU: Bringing Self-Adaptation to Microservices Architecture ... 77
  6.1 Overview of the GRU system .......................... 78
  6.2 GRU Architecture .................................. 81
    6.2.1 Cluster configuration ........................... 81
    6.2.2 Automatic resource management .................. 83
    6.2.3 Microservices Registration ...................... 83
    6.2.4 GRU-Agents Interactions ......................... 83
  6.3 GRU-Agents ........................................ 84
  6.4 System implementation ................................ 91
  6.5 Limitations ........................................ 93
  6.6 Conclusion ........................................ 94

7 Evaluation of the GRU prototype ......................... 95
  7.1 Case study ........................................ 96
  7.2 System Configuration ................................ 98
  7.3 Evaluation ........................................ 101
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.1 Test I: reactive test</td>
<td>101</td>
</tr>
<tr>
<td>7.3.2 Test II: bimodal test</td>
<td>104</td>
</tr>
<tr>
<td>7.4 Discussion</td>
<td>105</td>
</tr>
<tr>
<td>7.4.1 Analysis of GRU results</td>
<td>105</td>
</tr>
<tr>
<td>7.4.2 Comparison with the SELFLET Framework</td>
<td>118</td>
</tr>
<tr>
<td>7.5 Conclusion</td>
<td>120</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>121</td>
</tr>
<tr>
<td>9 Future Work</td>
<td>125</td>
</tr>
<tr>
<td>9.1 Future work related to the design of the tool</td>
<td>125</td>
</tr>
<tr>
<td>9.2 Future work related to the GRU prototype</td>
<td>126</td>
</tr>
<tr>
<td>Bibliography</td>
<td>129</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Distributed software systems are a fundamental part of the current IT scenario: the increasing importance and utilization of cloud computing and the advent of cyber-physical and pervasive systems pose new challenges to software engineers, who have to design and implement systems composed by a great number of elements interacting between each other in different ways.

Software is no more a monolithic unit working in isolation on a single machine providing services and functionalities to a small group of users: current software systems interact not only with millions of users that expect to have their requests satisfied quickly, but also with other smart devices such as smartphones or sensors to collect and analyze terabytes of data.

In order to deal with this scenario, software systems are now extremely complex and composed of thousands of pieces distributed on a large scale and communicating between them. Moreover, these systems work in a dynamic and unpredictable environment, since components may join or leave the system (e.g., virtual machines in a cloud environment, smart devices in a cyber-physical system, etc.), the number of users may change suddenly producing a peak or a drop in the workload, the failure of any part of the system can happen. The software system should be able to handle these
Chapter 1. Introduction

events and should guarantee to provide its services with the same level of quality.

For these reasons, the software system should be able to adapt to every change that happen in the context where it operates. However, due to the extreme complexity and dynamism of the depicted software system, manual adaptation by an operator would be nearly impossible; the software system should be able to adapt autonomously, reducing (or ideally eliminating) human intervention in the adaptation process.

1.1 Self-Adaptive Systems

The research on Self-Adaptive Systems is concerned with the design, analysis and evaluation of systems able to self-adapt, i.e., able to adapt themselves in respect to to changes in the environment where they are operating. Self-Adaptive Systems have been widely studied in the last decade, and very different solutions and approaches have been proposed (see Chapter 2 for further details).

Among the most popular ones, there is the Autonomic Computing model proposed by IBM [91]. An Autonomic System should present four characteristics, called the self-* properties: (i) self-configuration, which is the capability of the system or part of it to (re)configure itself according to the changes in the environment; (ii) self-healing, which regards the autonomous recovery from failures or at least the fault-tolerance of the system; (iii) self-optimization, which involves the autonomous optimization of resources according to some user policies; (iv) self-protection, since the system is potentially exposed to malicious software and users, it should be able to protect itself against possible threats.

Autonomic Computing is based on a feedback loop, called the MAPE-K loop (Monitor, Analyze, Plan, Execute over a Knowledge base) that is implemented in a component called Autonomic Manager, in charge of the self-adaptation process. The Monitor component should gather data on the status of the system and on the Managed Element. This data is then processed by the Analyzer component, producing higher level information exploited in the decision making process. The Planner decides if some action is needed on the system according to the results of the Analyzer and the data present in the Knowledge base. The Executor actuates the actions that are decided by the Planner component. The Autonomic Manager communicates to the Managed Application through Sensors and Effectors that are integrated inside the Managed Application itself (see Figure 1.1). The feedback loop and the interaction between the Adaptation Manager and the
1.2. Self-Adaptation in Distributed Systems

Managed Element enable the self-adaptation of the system.

In our research we adopted the presented MAPE-K feedback loop to enable the self-adaptation of the system.

1.2 Self-Adaptation in Distributed Systems

Self-Adaptation has been applied to different systems at different levels. In this thesis we focus on distributed systems. A distributed system is a set of computational elements that interact between each other and appear as a unique entity to the user. This kind of systems is very common in the cloud computing scenario, as well as in the Internet of Things field.

The application of Self-Adaptation to a distributed system is a very challenging task and some aspects need to be addressed:

- **Absence of a global knowledge.** Self-Adaptation requires the knowledge of the status of the system to make effective decision about the actions to actuate on it. However, the huge number of elements of a distributed system and their distribution on a large-scale makes difficult to share the global status of the system;

- **Changing and dynamic environment.** Distributed systems often present a high level of dynamism that cannot be controlled: components may join or leave the system at runtime changing their inter-
Chapter 1. Introduction

connections. The topology of the system itself may change, due to failures of some component or parts of the system. Self-Adaptation should be able to handle such dynamism;

• **System stability and resource consumption.** Self-Adaptation should be actuated taking into consideration the resources available to the system and avoiding to over or under-react to the changes that happen in the environment. The absence of global knowledge and the high dynamism can lead to the actuation of unnecessary optimization actions that may have as result the instability of the distributed system and the consumption of precious computational resources.

• **Unreliable network.** In order to effectively apply Self-Adaptation, the component in charge of the self-adaptive process should communicate to the different distributed components, both to gather data through sensors or to actuate actions through effectors. However, the network is unreliable, and issues like network delays, loss of packages, limitations on the bandwidth or network partition may compromise the communication process. Self-Adaptation must be able to operate even with unreliable network.

Self-Adaptation has been applied in various ways to distributed systems; in particular we can classify the various approaches considering the location of the self-adaptive process: there are *centralized* approaches, where one element is in charge and manage the others [69, 71, 145]; *decentralized* approaches, where all elements are peers and the self-adaptive behavior of the system emerges from the interaction of all the elements [107, 155, 156]; *hybrid* approaches, that are a mixture between the previous two [32, 79, 88, 151].

In this thesis we focus on decentralized approaches for the self-adaptation of a distributed system. Decentralization presents characteristics that are important when dealing with distributed systems; among them we can point out the following:

• **Scalability.** Decentralized systems can be scaled with ease. The absence of a central component makes easy to remove or add computational resources, scaling the system virtually to an infinite number of components. The spreading of the management component logic to every node avoids the bottlenecks that may arise in a scenario with millions of elements to manage, both for communication and computational resources. The main issue is related to the management of
1.3. Research goals

We motivated the need for the application of Self-Adaptation to distributed systems and highlighted the complexity of such task. The use of a decentralized system can provide some advantages compared to a centralized one, as we pointed out in the previous section.

Starting from this consideration, our research is based on the following goals.

**Research Goal 1: To assess the utility of decentralized self-adaptation by experimenting with it**

The starting point of this thesis is the evaluation of a decentralized self-adaptive system through a realistic case-study to understand its effectiveness and utility. Literature provides few examples of this kind of evaluation, focusing on toy examples and simulations.

The analysis of the obtained results can help to understand the validity of the choices at the base of the design of the Self-Adaptive System, and address possible issues and limitations, filling the gap between the theory behind these systems and their concrete application by practitioners.

**Research Goal 2: To design a software tool to enable transparent adaptation in Complex Systems**

The application of decentralized self-adaptation to an external distributed system that has already been deployed is not a trivial task. However, it
Chapter 1. Introduction

is important for the adoption by practitioners, to be able to apply Self-Adaptation to their distributed systems without complex and difficult modifications. For this reason, the study of a software tool able to introduce self-adaptive capabilities inside a distributed system in a transparent way is fundamental.

Research Goal 3: To implement a prototype of the designed tool, and to validate it through its application to a concrete application domain

The development of a prototype of the designed tool is fundamental to highlight possible technological issues that may arise during its implementation. The developed prototype can be applied to a concrete and realistic application domain to validate its design and its effectiveness through a robust evaluation. This evaluation can help in the identification of its issues, and provide insights on the directions for future researches.

1.4 Contributions

The contributions provided with our research aim to address the research goals defined in the previous section. In this section we present each contribution to every defined research goal.

Contribution to Research Goal 1: Evaluation and analysis of the SELFLET Framework, a decentralized self-adaptive system

The first research goal has been addressed improving and evaluating an existing decentralized self-adaptive framework called the SELFLET Framework [42]. Through this evaluation we can understand the effectiveness and utility of decentralized Self-Adaptation applied to a distributed system. We improved the system introducing new algorithms for the decision making process and through the fine tuning of the parameters to obtain better performance. We created a concrete case study based on the domain of video-on-demand services and implemented it using the framework itself. We executed an evaluation of the system through a deployment in the Amazon cloud computing platform. We did an analysis of the design choices at the base of the framework itself, highlighting the advantages and disadvantages of each choice. This analysis helped us to understand the possible limitations of the SELFLET Framework, such as the need to learn the framework and the constraints on the technology stack of the application.
1.4. Contributions

Contribution to Research Goal 2: Design of a software tool to enable the transparent decentralized self-adaptation in distributed systems

We designed a tool that overcomes the limitations emerged from our analysis, and that can bring decentralized self-adaptation to an external distributed system in a transparent way. We propose the technologies that are best suited for its implementation, the Microservices Architecture pattern and Docker containers, providing also an overview of the literature about their use in the context of Self-Adaptation.

Contribution to Research Goal 3: implementation and application of GRU – a prototype of the designed tool – to the domain of applications built using the Microservices Architecture pattern and deployed in Docker containers

We implemented a prototype of the designed software tool called GRU, and applied it to cloud-based applications developed using the Microservices Architecture pattern and deployed inside Docker containers. GRU exploits Docker containers to introduce self-adaptive capabilities inside the application in a transparent way. The Microservices Architecture is a recent trend in cloud-based application development adopted by the major IT companies, and the evaluation of GRU in this context provides a strong validation of our approach.

Publications

In this Section we present all the work published as a result of our contribution to the aforementioned research goals.


Authors: D. Ardagna, N.M. Calcavecchia, L. Florio, E. Di Nitto, D.A. Tamburri

Venue: IEEE Transactions on Services Computing (under revision).

Summary: This paper presents the SELFLET Framework, covering its architecture and its implementation. The adaptation loop is described in details, as well as all the algorithms related to the implemented adaptation policies. The paper contains also the evaluation of the framework using a concrete case study.

Contribution: My contribution to this paper regards the improvement of the SELFLET Framework through the imple-
Chapter 1. Introduction

Implementation of new algorithms and the fine tuning of parameters. I executed also the evaluation part, preparing the case study and deploying the system on the Amazon Cloud infrastructure. I contributed to the writing of the paper.

Usage in this thesis: This work is the base of the chapters related to the SELFLET Framework, i.e. Chapter 3 and Chapter 4. The analysis presented in Chapter 5 is also based on this work.

Contribution to research goals: This paper contributes to Research Goal 1, providing the evaluation of a decentralized self-adaptive system through a concrete case study. The analysis of the SELFLETs has been done starting from the results obtained in this work.

[Paper B] Decentralized Self-Adaptation in Large-Scale Distributed Systems

Authors: L. Florio


Summary: This paper is an overview of my Ph.D. research, describing the background, the goals we want to achieve and a plan of the research.

Contribution: This paper is the description of my Ph.D. research project, so I contributed in its definition and the writing of the paper.

Usage in this thesis: There is no a specific usage in this thesis, however this paper constitutes the general view about my research.

Contribution to research goals: This paper does not contribute directly to any Research Goal, however it played an important role in their definition.

[Paper C] GRU: an Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures

Authors: L. Florio, E. Di Nitto

1.4. Contributions

**Summary**: This work presents the initial design and the ideas behind GRU, the software tool to introduce decentralized self-adaptation in microservices applications deployed in Docker containers. An initial evaluation of the first prototype is provided.

**Contribution**: My contribution to this work regards the idea and the design of GRU, its initial implementation as well as its initial evaluation. I contributed to the writing of the paper.

**Usage in this thesis**: This work is the foundation of Chapter 5 and Chapter 6 where – after the analysis of a decentralized self-adaptive system – we present the idea of an external software tool to bring Self-Adaptation to a distributed system in a transparent way.

**Contribution to research goals**: Research Goal 2 is addressed in part with this work, since it is the result of the analysis of a decentralized self-adaptive system and present the idea for the design of a software tool to bring self-adaptive capabilities to a Distributed System in a transparent way. This work presents also an initial implementation and evaluation of the prototype of the tool – i.e., GRU – so its contribution is related in part to Research Goal 3.

---

**[Paper D] GRU: Autonomic Microservices Management with GRU**

**Authors**: L. Florio, E. Di Nitto, D. A. Tamburri

**Venue**: IEEE Transactions on Cloud Computing (under revision).

**Summary**: This paper presents the design and implementation of GRU, the software tool able to make self-adaptive an application deployed in Docker containers. We describe its architecture, the implementation and we evaluate the tool with a concrete case study.

**Contribution**: I designed and implemented GRU and fine-tuned its configuration. My contribution regards also the implementation of the concrete case study using the Microservices Architecture pattern and Docker containers, as well as its deployment in PoliCloud – the private cloud of Politecnico di Milano – and its evaluation.
Chapter 1. Introduction

Usage in this thesis: Chapters 6 and Chapters 7 are completely based on this work.

Contribution to research goals: Research Goal 3 is addressed by this work. GRU is the prototype of the software tool to bring self-adaptive capabilities to an external Distributed System in a transparent way that we implemented and evaluated through a concrete case study.

1.5 Structure of the thesis

The thesis is based on the following structure.

Chapter 2 provides an overview of the state of the art in Self-Adaptive Systems, as well as their application to cloud computing. This chapter can help the reader in positioning our work among the studies done by academic research about Self-Adaptive Systems.

Chapter 3 presents the SELFLET Framework and highlights the contributions to the framework. We describe its architecture, focusing on its component and on the adaptation loop of the system.

The evaluation on the SELFLET Framework is described in Chapter 4. We describe in details the case study implemented as well as the configuration of the system for the tests performed on the Amazon cloud computing platform. We present the results of the evaluation and discuss them.

Chapter 5 is dedicated to the analysis of advantages and disadvantages of the SELFLET Framework, and the definition of a new tool to apply self-adaptive capabilities to a distributed system in a transparent way. We point out the limitations of the framework, and present the design of a tool that aims to overcome such limitations. We propose the application of this tool to the domain of microservices applications deployed in Docker containers, and provide a brief introduction to these technologies as well as the description of the effort done by research in the application of Self-Adaptation to these systems.

We implemented the proposed tool, creating a prototype called GRU. This prototype is described in details in Chapter 6. We show the architecture of GRU focusing on its configurable parts and its internal components. We also provide an example of deployment of a GRU system and we briefly describe its technical implementation.

Chapter 7 presents our evaluation of the GRU tool using a concrete case study. We analyze and discuss the results obtained with our tests, comparing GRU to the SELFLET Framework when possible.
1.5. Structure of the thesis

Finally, Chapter 8 summarizes the results obtained with our work and Chapter 9 illustrates possible future works that can be the subject of further studies on the topics addressed in this thesis.
CHAPTER 2

State Of The Art

In this chapter we provide the state of the art related to Self-Adaptation and its application to the context of cloud computing. The concepts introduced here are required to better understand the work presented in this thesis.

Firstly we describe **Self-Adaptation** in Section 2.1, analyzing the characteristics of a self-adaptive software and presenting the different ways it is implemented in literature. Self-Adaptation is the core of our research, so it is necessary to understand the work that will be presented in the next chapters and to contextualize it according to the current state of the art in this field.

Section 2.2 describes **cloud computing** and the research efforts to solve some of its issues using Self-Adaptation. Cloud computing is the context where we applied our research on Self-Adaptation.

We summarize our conclusion in Section 2.3, highlighting the contribution of this thesis with respect to literature.

### 2.1 Self-Adaptation

Self-Adaptation is the capability of a system to autonomously adapt itself to the variation of the environment where it operates \cite{91,113}. The adap-
Chapter 2. State Of The Art

Self-Adaptation happens adjusting parameters and changing the components of the system. Self-Adaptation comprehends different self-* properties, like self-configuration, self-healing, self-optimization and self-protection \[\text{[84, 91]}\]. In order to provide such self-* properties, a Self-Adaptive System (SAS) should be able to monitor itself as well as the environment where it operates, to detect changes and variation both in its internal state and in the state of the environment (its context); this means that the SAS should be self-aware and context-aware \[\text{[78, 131, 132]}\]. SASs are usually composed of two elements: the managed application and the adaptation logic \[\text{[156]}\]. The managed application is the component(s) of the system that is adapted by the adaptation logic, that can be able to adapt itself as well as the context where the application operates. The adaptation logic is usually divided in four components that monitor the environment, analyze the data gathered, plan the actions to actuate, and execute such actions. This set of components is commonly referred as the MAPE loop \[\text{[91]}\].

2.1.1 Self-Adaptation properties

Literature provides the classification of SASs according to some properties related to the Self-Adaptation mechanism, such as the time of the Self-Adaptation, the level where it is applied, its integration inside the system, the strategy used to decide how to adapt and the location of the adaptation logic \[\text{[96, 125, 131]}\].

Time

The time property refers to when the adaptation should be actuated and is usually divided in reactive and proactive Self-Adaptation \[\text{[75]}\].

Using a reactive approach, the system self-adapts when some event triggers the self-adaptive process (e.g., the value of a variable goes over a defined threshold, a loss in performance, etc.) \[\text{[91]}\].

The proactive approach makes the system able to identify the need for Self-Adaptation before the event occurs, anticipating a loss in performance or the future need of new resources \[\text{[104]}\].

Despite the proactive approach can be more effective it is more complex to implement, requiring the implementation of predictive algorithms to anticipate the future need for self-adaptive actions.

Level

Self-Adaptation can be actuated at different levels inside the SAS \[\text{[75, 125]}\].
2.1. Self-Adaptation

The **resources** of the system such as hardware resources (i.e., CPUs, memory, storage, etc.), virtual machines, containers, physical devices, can be altered by the self-adaptive process, e.g., adding or removing them as needed \[61, 112\]. On top of the resources runs a **system software** like an Operating System or a Middleware, that can be reconfigured through self-adaptive actions \[118, 141\]. The managed **application** is deployed over the system software and can be composed by a single component or distributed in a network of different devices. The elements composing the system should interact between them through a **communication system** that comprehend both the **physical infrastructure** and the logical **communication** (i.e., the communication pattern adopted) and that can be adapted according to the context (e.g., switching from a wireless protocol to another) \[31, 123\]. The **context** can trigger self-adaptation mechanism or the context itself can be adapted by the SAS if it has actuators to interact with it \[28, 36, 132\].

Self-Adaptation may operate on multiple levels on the system and is not confined on a single one. However, coordination problems need to be handled, especially when multiple self-adaptation managers are involved \[51, 56\].

**Strategy**

Self-Adaptation can exploit different strategies to decide the adaptation action to actuate. Such strategies comprehend **models**, **rules** and **policies**, **goals** or **utility functions** \[97\]. These strategies can be combined in the decision making process of the adaptation logic.

Model-based Self-Adaptation uses **models** of the actual state of the system and the desired state to decide the actions to actuate. The adaptation logic creates and analyze the models of the current status of the system, then from this analysis a plan is created in order to bring the system to a desired state \[76, 83, 90\].

**Rules** and **policies** can be used to trigger self-adaptive actions and to determine how the SAS should react to specific events \[91, 129\]. Usually policies and rules are defined at design-time, making the system less flexible to the various situations that may occur.

**Goals** can be used to drive the decision about the plan to actuate, with the SAS aiming at satisfying some predefined goals \[40, 50, 103\]. It is worth noting that the goals can be contradicting and change over time, so the adaptation logic should be able to resolve the conflicts that may arise.

The use of an **utility function** can be exploited to select the best self-adaptive actions to actuate \[142, 143, 150\]. The goal of the SAS is to create
Chapter 2. State Of The Art

a plan that maximize the utility function taking in consideration some variables and costs. The drawback of this approach resides in the difficulty to define a proper and effective utility function.

Integration

The integration of the adaptation logic usually follows two opposite approaches, the internal approach or the external one.

The SASs using the internal approach integrate the adaptation logic inside the managed application [42]. The application logic is part of the managed application itself and can internally monitor the status of the system and directly actuate modifications on it. This approach is effective for local adaptations, but may lead to issues in scalability and maintainability.

The use of the external approach implies the separation between the managed application and the adaptation logic that communicate between them using an interface, making the SAS modular and more maintainable [131]. The interaction between the adaptation logic and the managed application happens through the use of sensors and effectors [91]. Sensors are used to monitor the status of the managed application and gather the data needed to understand the status of the system. Effectors provide a way to operate on the managed application and enable the Self-Adaptation.

Location

Not only the integration of the adaptation logic may vary in different SASs, but also its location. In particular, the adaptation logic can be implemented using centralized, decentralized or hybrid approaches.

In centralized SASs the adaptation logic is implemented inside a management layer that is in charge of all the autonomic operations (e.g., [69, 71, 145]). This layer is centralized, making possible to maintain a global state of the system. This solution can facilitate the process of analyzing and planning, however the overhead for communication can be relevant and the centralized manager can be a single-point-of-failure for the system.

Decentralized SASs don’t have a central management layer and the adaptation logic is spread more or less uniformly over all elements (e.g., [107, 155, 156]). This solution is suitable in large-scale distributed systems composed of thousands of components. The information about the status of the system is limited, so it is not possible to have a global state of the system and the self-adaptive actions may be less effective.
2.1. Self-Adaptation

Hybrid SASs can achieve self-adaptation using a combination of centralized and decentralized approaches (e.g., [39, 92, 151]). The adaptation logic is structured as a hierarchy of adaptation layers, so that the lower layers interact directly with the managed application guaranteeing a timely adaptation, while higher level layers have a more global vision and can plan adaptation at a longer time scale. The drawbacks of this approach is the complexity in the design and management of hierarchies with multiple layers.

2.1.2 Self-Adaptation implementation

Self-Adaptive Systems have been implemented in various ways in literature. In this section we want to provide an overview of some of the different implementation approaches, focusing on the ones that are most relevant to our work.

Table 2.1: Self-Adaptation implementations.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Time</th>
<th>Level</th>
<th>Strategy</th>
<th>Integration</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Loop</td>
<td>React</td>
<td>Sys / Res</td>
<td>Policies / Goals</td>
<td>External</td>
<td>All</td>
</tr>
<tr>
<td>Multi-Agent</td>
<td>React</td>
<td>App</td>
<td>All</td>
<td>External</td>
<td>Dec</td>
</tr>
<tr>
<td>Bio-Inspired</td>
<td>React</td>
<td>App</td>
<td>Utility Functions</td>
<td>External</td>
<td>Dec</td>
</tr>
<tr>
<td>Service-Oriented</td>
<td>React</td>
<td>App</td>
<td>Models</td>
<td>External</td>
<td>All</td>
</tr>
<tr>
<td>Architecture-Based</td>
<td>React</td>
<td>App / Res</td>
<td>Models</td>
<td>External</td>
<td>All</td>
</tr>
</tbody>
</table>

Table 2.1 summarizes the different properties of the described implementations to provide a clear view of their differences. The Table shows the most common choices about the properties in each implementation. The term All means that there is not a choice that is preferred to the others, but that are all commonly used. Proactive self-adaptation is possible in all the implementations, but, due to the complexity that derives from its adoption, a reactive (React) approach is usually preferred. The most common level of adaptation is application (App), followed by resources (Res) and system (Sys). The strategy varies according to the implementation, and the integration is usually external. There is not a location that is preferred to the others, except in the Multi-Agent and Bio-Inspired implementations that are usually decentralized (Dec).

For a more exhaustive overview on the various approaches the reader is referred to [96, 131].
Chapter 2. State Of The Art

Control Loop approaches

The use of a control loop to enable self-adaptive capabilities is a common approach in Self-Adaptation [65, 66, 116]. The feedback loop is widely used to implement reactive Self-Adaptation: it allows the reaction to changes in the external or internal status of the system. In general, a feedback loop starts gathering the data about the status of the system; then these data are analyzed to obtain a high-level view of the system; starting from the high-level view, a decision phase chooses the actions that should be executed; the loop ends with the actuation of the chosen actions.

The most common implementation of the feedback loop is the MAPE-K loop (Monitor, Analyze, Plan, Execute, over a Knowledge base) envisioned in the Autonomic Computing manifesto [91]. The traditional MAPE loop is enriched with a knowledge base that is exploited to makes the decisions in the planning phase. The MAPE-K loop has been used as a reference for the implementation of several SASs (e.g., [63, 85, 94, 130]).

SASs based on a feedback loop are usually reactive, despite a proactive approach could be possible with the use of a feedforward loop. Using the MAPE-K implementation, the adaptation logic is often external to the managed application and can be decentralized. We use the MAPE-K loop to enable the self-adaptation of the system. However, the knowledge base in our loop is not complete, but derived from a partial knowledge of the status of the system.

Multi-Agent approaches

An Agent is a software element that is able to act autonomously according to the status of the environment and its internal logic. Agents can be combined in a system in order to reach a common goal through communication and coordination, forming a Multi-Agent System (MAS) [64, 148]. MAS present features that can be exploited for the development of a SAS: they are designed to be distributed and decentralized, are flexible and can deal with a dynamic environment where components can join or leave the system at any moment [152, 154]. These reasons led to the study of MAS for the implementation of self-adaptive systems [55, 93, 162].

Unity is an example of how to take advantage of the MAS paradigm to create a decentralized architecture based on multiple interacting self-adaptive agents called autonomic elements [142]. The system is able to self-optimize the resources allocation in a dynamic multi-application environment. In Unity utility functions are used to allow the system to manage itself. Resource-level utility functions are computed by each auto-
2.1. Self-Adaptation

A decision-making agent within an application environment, using information based on service-level utility functions of each application. Resources are allocated by a Resource Arbiter element, which computes a globally optimal allocation of resources based on the resource-level utility functions provided by the agents. Unity is focused on the self-optimization of resources allocation and on the self-organization ("goal driven self-assembly") of agents. The policies of the system are high-level utility functions provided by the user.

MAS are often studied in relation to the concept of Self-Organization, a particular kind of Self-Adaptation [115, 134, 161]. Self-Organization is defined as a mechanism or a process which enables a system to change its organization without explicit command during its execution time [134]. Agents can organize autonomously to manage services and resources, self-adapting the system according to its needs [62, 81]. Complete overviews about Self-Organization approaches and mechanism can be found in literature [115, 161].

Self-Adaptation using MAS is usually external and decentralized, since there is not a centralized unit controlling all the agents in the system. The self-adaptive process can be proactive as well as reactive, and the adaptation is most common at the application level, despite it can happen potentially at any level. The use of goals to plan the self-adaptive actions is common in MAS, however also policy/rules and utility functions can be used to drive the self-adaptive process. The focus in this kind of implementation is in the planner component of the MAPE loop, since the use of several agents makes the planning process not trivial. The approach we propose is based on MAS, due to our focus on decentralized self-adaptation. In particular, our solution is based on the organization of agents in small groups of communicating peers to reduce the communication overhead.

Bio-Inspired approaches

Bio-Inspired approaches are based on algorithms derived from biological and natural systems. These systems are usually composed by a large number of components that interact between them without explicit coordination and that share only a limited knowledge. The collective behavior of the system emerges from the behavior of each single unit: this property is called emergence [41, 57, 74]. The study of this collective systems led to the adoption of Bio-Inspired approaches for the engineering of systems in the computer science field [37, 59, 114]. Several biological mechanisms have been adopted for self-organization and for the engineering of SASs; among them foraging, quorum sensing, consensus, stigmergy, the human immune system, and many others [24, 98, 99, 106].
Chapter 2. State Of The Art

Clonal plasticity is one of the most recent works on Bio-Inspired approaches [108]. The authors present a method to obtain decentralized adaptation in Multi-Agent Systems that does not present an explicit MAPE loop, but that is inspired by plants, in particular two capabilities: (i) phenotypic plasticity, that means that the environment can influence the traits of the plant (height, leaf dimension, etc.); (ii) clonal reproduction, that is the ability to reproduce itself through cloning. Clonal plasticity combines these two features, so in the proposed solution an individual can reproduce itself by cloning and altering its behavior on the basis of the data coming to the environment. The system also uses a plastic memory to remember the previous adaptations and repeat them if they had a positive reward, or discard them if they were not effective. The approach to self-adaptation is interesting and original, but the evaluation is limited to a simple simulation example, without providing a complete and realistic validation.

Bio-Inspired approaches shares the properties of MAS-based approaches. However, since the emergence of the global behavior of the system cannot be completely controlled, the planning phase requires a particular attention. We exploit the emergence property to avoid an explicit coordination between the agents of our system, reducing the possible instability that may derive from emergence through the use of a probabilistic algorithm for the planning phase.

Service-Oriented approaches

SASs based on the Service-Oriented approach are applied to systems composed of services (i.e., independent and autonomous software entities that provide a specific functionality) [35]. Application composed of services usually are organized according to a Service-Oriented Architecture [126].

The MUSIC Framework provides a platform to build service-oriented SASs using a model-driven approach [127]. The SASSY Framework allows to automatically generate software architectures according to Quality-Of-Service (QOS) constraints imposed by stakeholders and the actual environment. The SASSY Framework operates both at design time and at runtime, reconfiguring the architecture if needed [102]. Another framework where self-adaptation is driven by QOS constraints is the MOSES Framework: it enables the self-adaptation of a service-oriented system implementing different adaptation mechanism to face the various operating environments and the possibly conflicting QoS requirements of several concurrent users [49].

GoPRIME is a middleware for the autonomic service assembly based on PRIME, a previous work of the same authors [47, 48]. GoPRIME is
2.1. Self-Adaptation

fully decentralized and is designed for the adaptive self-assembly of distributed pervasive systems. In general, GoPRIME operates on distributed systems composed of set of peers that cooperates between them to accomplish a task. Services are able to perform a specific task, but each service could depend on services provided by another one. GoPRIME is able to manage the system in order to select the correct assembly that fulfills global non-functional requirements. The core of GoPRIME is a gossip protocol for information dissemination and decentralized decision making.

Service-Oriented approaches are usually reactive and the decision is often based on models (e.g., [102, 127]). The adaptation happens at the application level (the services). We apply our self-adaptive approach to applications based on services, providing several self-adaptive capabilities to adapt the applications to the changing workload they can face.

Architecture-Based Approaches

Architecture-Based Self-Adaptive Systems exploit Software Architectures to enable Self-Adaptation [2, 31, 153]. A Software architecture provides the global system level perspective, and facilitates application programmer with properties and behavioral abstractions to work with. Moreover, the architecture enables the understanding of system’s topological and functional constraints at the higher level and provides a better way to ensure the validity of system with changing needs. The goal of architecture-based adaptation is to minimize human intervention for managing the system in a way that system should be able to organize itself according to the architectural specification.

Rainbow is probably the most famous framework for architecture-based Self-Adaptation [71]. Rainbow implements an autonomic manager composed of the system-layer infrastructure, the architecture-layer, the translation infrastructure and a system-specific adaptation knowledge. Through a distributed set of probes and gauges data are gathered from the application. The centralized architecture evaluator analyze the data to detect problems and the adaptation manager decides the best action to actuate, that is then executed by the effectors. Rainbow has been applied to an industrial system to improve its self-adaptive capabilities [44, 45]. The industrial system, a middleware used to monitor and manage networks of devices, had already self-adaptive capabilities but has been improved making it more flexible and maintainable.

SASs exploiting Software Architectures usually use models to represent the architecture of the systems [46, 67, 68]. These models can be used with policies, strategies or constraints to reason about the structure of the sys-
Chapter 2. State Of The Art

Adaptation logic is external to the application to manage. Despite our approach is not Architecture-Based, it is relevant in our research because it is one of the few examples of the application of self-adaptation in a concrete environment. We want to move in that direction, providing a self-adaptive system that can be applied seamlessly to an industrial software without requiring its modification.

2.2 Cloud Computing

Cloud computing is defined as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [101].

Cloud computing providers offer resources and services on demand; these services can be grouped in three categories [101, 163]:

- **Infrastructure as a Service** (IaaS): the cloud provider offers infrastructural resources, that usually are in the form of Virtual Machines (VM);

- **Platform as a Service** (PaaS): the user has access to a software platform that can be an Operating System or a development framework;

- **Software as a Service** (SaaS): SaaS refers to on-demand applications over the internet that are accessible by the user.

The deployment of a cloud infrastructure can happen in four different ways, that vary according to how the resources inside the cloud platform are shared. In a **Private Cloud** the resources are shared only inside the organization that deployed the cloud infrastructure; using **Community Cloud** two or more organizations share the resources of a common cloud infrastructure; **Public Cloud** is totally open and the resources can be accessed by any user upon request; **Hybrid Cloud** is a mixture of the previous defined models: the cloud infrastructure presents a part of the resources that are private and a part that are public.

Cloud computing has been widely adopted for the deployment of web applications, allowing the dynamic scaling of such applications through on-demand resource provisioning that led to a reduction of costs for IT companies [27, 163]. Complete overviews about cloud computing and its characteristics can be found in [27, 101, 163].
2.2. Cloud Computing

Cloud computing is a good application domain for Self-Adaptation, which has been applied in most cases to address a specific problem, like load-balancing, fault-tolerance, and resource management. We want to contribute to the research in this field, proposing a self-adaptive approach that is decentralized and can provide self-adaptive capabilities able to handle all the most common issues that an application deployed in a cloud computing environment can face. We want to evaluate our approach using a realistic case study deployed in a cloud infrastructure. This kind of evaluation is not common in literature, where the majority of the proposed solutions have been validated using simulation, but we believe it is fundamental for the concrete adoption of Self-Adaptation in cloud computing.

2.2.1 Cloud Applications Self-Adaptation

Despite its success in the IT industry and its advantages, Cloud Computing presents some challenges like dynamic workloads, availability, resource management and scaling [120, 163]. Self-Adaptation has been applied to cloud computing in order to face such challenges, providing solutions to bring complete self-adaptive capabilities to a cloud system or to address specific issues (e.g., [56, 63, 107, 139]).

The Design of an adaptive system deployed in the cloud has been studied in [165]. The core of the adaptation process is the optimization of an objective function that is derived on the basis of non-functional requirements goals. The proposed solution uses control points, i.e., the artifacts in the system that are modified at runtime to enable changes in the system, to actuate the self-adaptive process. The adaptation logic is implemented with the traditional MAPE-K loop, that makes use of a search-based algorithm to determine the changes to operate on the control points in order to optimize the value of the objective function.

MODAClouds project applies a model-driven approach to the design and execution of applications in multiple clouds [26]. The project can help the developers to design, develop and deploy a software in a cloud-agnostic way, so that is possible to exploit multiple clouds at the same time. The runtime adaptation mechanism allows to react to changes in the context, such as variations in performance, requirements, etc., enabling the dynamic re-deployment of the applications and its components with a different configuration.

The Autonomic Cloud is proposed in the context of the ASCENS project that aims to provide tools and methods to develop software ensembles [157]. The proposed approach is based on autonomic systems composed by Ser-
vice Components, individual building blocks combined in a dynamic manner to form service component ensembles. Service Components are the basic building blocks of the autonomic system and can be reactive or proactive and communicate between them in various ways. Service Components can be managed by an Autonomic Manager Service Component that provides an external and explicit autonomic feedback loop. These components can be composed using different patterns to best fit various use cases [121]. These concepts are applied to the cloud computing scenario proposing the Autonomic Cloud, a solution for cloud self-management at the system level [100]. The system is based on a peer-to-peer architecture where each Service Component can communicate with the others to adapt to the changing condition of the environment, but in certain conditions (failure of a large part of the system) the system can change its topology electing a centralized Autonomic Manager Service Component to handle the situation. Once the recovery is complete, the system can switch back to the peer-to-peer pattern.

The problem of the dynamic and changing workloads that a cloud application could face has been addressed using various techniques for the autonomic load-balancing [122, 164]. This problem has been addressed also with bio-inspired and self-organizing approaches [95, 135, 160]. In particular, Mycoload uses a self-organized load-balancing algorithm to load-balance the load in overlay-decentralized service networks [147]. The system is an evolution of Myconet [138], a bio-inspired model for peer-to-peer overlay topology which is based on super-peers, and is able to self-organize a network of service nodes and balance the load among nodes with different services.

Self-healing techniques have been used to deal with fault-tolerance in cloud environments and to guarantee availability of the deployed application [54, 70, 105]. The cost of self-healing is terms of energy consumption is analyzed in [149], where it is presented a protection and recovery mechanism for cloud infrastructures. In [29] The Self-Healing of a cloud infrastructure is operated through the use of a Multi-Agent System able to handle failures through the analysis of the state of resources and the execution of Checkpoint/Replication strategies or migration techniques. Failure prediction and fault localization are the target of [159], where it is proposed a proactive approach based on a combination of data analytics and machine learning techniques.
2.2.2 Resource Management Self-Adaptation

Self-adaptive techniques are used in Cloud Computing platforms especially to manage resources and to allow the scaling of computing instances and services [137].

Cloud providers offer integrated system for the auto-scaling of active instances that can be totally transparent to the user or that can be configured using policies and rules: **Google Cloud Platform** takes care of the scaling of the system, which is completely transparent to the user [11]; **Amazon Web Services** uses auto-scaling policies defined by the user [2]; **Microsoft Azure** allows the user to control the auto-scaling system defining rules [19].

Literature provides more sophisticated methods for the auto-scaling of the cloud infrastructure, using model-driven, predictive, probabilistic or MAS-based approaches [60, 87, 111, 128].

**Multi-Agent Systems** are exploited for the automatic and dynamic resource provisioning in cloud infrastructure in [25]. The Multi-Agent System is used to manage the resources of the cloud provider, scaling them dynamically in order to satisfy the quality-of-service requirements of the customers determined by the Service Level Agreement. The solution is based on local utility agents assigned to each customer that performs local optimization and request the resources to a centralized global utility agent that manages all the resources of the cloud provider datacenter and acts as a central broker.

This solution is centralized like others proposed in literature (e.g. [73, 86, 136]). However, it is possible to find also interesting decentralized approaches [38, 43, 158].

**Mycocloud** uses a decentralized and self-organizing method for the service placement in a cloud environment composed of heterogeneous resources, enabling service elasticity [62]. The approach is bio-inspired and it is based on a self-organizing network of nodes connected via a peer-to-peer overlay.

A complete overview of the autonomic resource management in cloud environments can be found in literature [137].

2.3 Conclusion

This chapter presented the state of the art in the context of Self-Adaptation and its application to cloud computing. The concepts introduced here constitute the background information the rest of the thesis relies on.

In Section 2.1 we presented an overview of **Self-Adaptation**, analyzing its properties and the various ways it has been implemented in literature.
Chapter 2. State Of The Art

This overview can help the reader to better place our work in relation to the current research on Self-Adaptive Systems, and to understand the choices made in our implementations (both the SELFLET Framework in Chapter 3 and GRU tool in Chapter 6).

Section 2.2 quickly described the cloud computing scenario and motivated the need for Self-Adaptation to address its issues. Several works that represent the current state of the art have been described, ranging from complete solutions that would lead to an autonomic cloud to works that focus on single topics like changing workload, availability, and resource management. The work we present in Chapter 3 and Chapter 6 both operate in this context, providing different solution for the application of Self-Adaptation to the cloud computing scenario.

We can conclude that literature presents several approaches related to Self-Adaptation that differ on the implementation of the properties related to the self-adaptive process. The context of cloud computing is a perfect fit for the application of Self-Adaptation, that has been used especially to address the problem of resource management. Despite the effectiveness of all the approaches and algorithms implemented in the various systems, there is very little attention to the aspects related to the concrete adoption of Self-Adaptation in realistic contexts. The majority of works have been tested using simulations and toy-examples, without concrete case studies that can provide a robust evaluation of the Self-Adaptive System. The problem of integration of Self-Adaptation inside an external deployed system has received very little attention. However, we believe that these aspects – i.e., the evaluation through a concrete case study, and the integration in an external deployed system – are fundamental aspects for the adoption of Self-Adaptation by practitioners. This thesis moves in that direction, focusing not only on algorithms for Self-Adaptive Systems, but on the study of how to effectively enable adaptation capabilities in an external distributed system that has not been designed to integrate them.
CHAPTER 3

The SELFLET Framework

In this chapter we describe the main features of the SELFLET Framework, a decentralized self-adaptive framework to build distributed service-based applications. The SELFLET Framework is a Multi-Agent based system, where every SELFLET is an agent that implements an internal MAPE-K loop for the decision making process. The system is decentralized and every SELFLET decides locally the autonomic action to actuate on the basis of its internal state and the state of a set of other SELFLETS. The global adaptation of the whole system emerges from the local decision of every SELFLET, implementing the concept of emergent behavior typical of bio-inspired approaches (see Chapter 2). The adaptation is based on autonomic policies. The SELFLET Framework uses an internal approach: the user should develop the application using the framework, so the management layer is inside the application itself. Autonomic actions involve the allocation of new resources and modification of the behavior of the implemented services, so the adaptation happens at the resources and application level.

The SELFLET Framework has been presented in a previous work focused on its utilization for the design of a service-based application [42]. In this thesis we evaluate the self-adaptive capabilities of the SELFLETS using a realistic case study. This chapter and the following one extend the
Chapter 3. The SELFLET Framework

work presented in [42], providing several contributions:

- **C1: Dynamic threshold.** We implemented a system to dynamic set the utilization upper bound of a SELFLET (described in Section 3.3). In this way it is not necessary to define it at design-time, but the system can compute it dynamically at runtime;

- **C2: Implementation and improvement of different action generation algorithms.** We implemented algorithms for the Action Generation phase that were defined only theoretically, and designed the Algorithm 3 used for the Change Implementation action;

- **C3: Fine tuning of parameters.** Testing the system several times, we defined the set of parameters for the configuration of the system that provides the best performance on the experimental benchmark;

- **C4: Design and implementation of a realistic case study.** The SELFLET Framework has been evaluated only in simulation and on toy examples. We developed a realistic case study that we implemented using the framework itself to provide a robust evaluation of its capabilities;

- **C5: Evaluation of the framework in a cloud environment.** We evaluated the system using the aforementioned case study, deploying it in a cloud environment composed of 50 nodes, and proving its effectiveness in a realistic and complex environment.

In this chapter we focus on the design and implementation of the SELFLET Framework, presenting its architecture and the adaptation loop used for the decision making process. The description of the SELFLET Framework has been presented in [42], but it is needed to understand the contribution provided to the framework itself. Chapter 4 is dedicated to the description of the case study and the evaluation of the framework.

In particular, this chapter is structured as follows: an overview of the system is presented in Section 3.1. Section 3.2 presents the general architecture of the system and its main elements, i.e. Services, Behaviors and Autonomic Policies; Section 3.3 describes the adaptation loop implemented in the SELFLETs, presenting the algorithms for the generation of the autonomic actions and the action selection. The internal architecture of a SELFLET is described in Section 3.4 where an overview of the internal components of the SELFLET agent is provided. Section 3.5 concludes the chapter.
3.1 Overview

This section provides an overview of a SELFLET application. Details about the concepts and the internals of a SELFLET are described in the following sections.

A SELFLET-based system is composed by many SELFLETS – i.e. computational nodes – spread over a logical network that have the same conceptual model and architecture, similarly to a peer-to-peer system. Moreover, each SELFLET has a unique ID that is used for referring to a specific SELFLET. Each SELFLET acts as a container for some services defined by the user through the framework.

The communication happens through messages exchanged using a message broker. In order to avoid communication overhead in the case of a system composed of thousands of nodes, every SELFLET can communicate and exchange information with a subset of the total number of SELFLETS. The subset is different for each SELFLET and is called the Neighborhood of the SELFLET. SELFLETS are capable of organizing themselves in different Neighborhoods: the consequence is that every SELFLET have a different and partial view of the state of the system that depends on its neighbors. The number of elements that can be part of a Neighborhood is set by the user in the configuration of the system. When a SELFLET joins the system, it has to contact a component called Requests Dispatcher, which represents the entry-point of the application and is in charge of the management of the Neighborhoods. The Requests Dispatcher assigns to the new SELFLET a list of neighbors, that are chosen among the other SELFLETS whose Neighborhood is not full. As a consequence, the new SELFLET is added to the Neighborhood of its own neighbors. In order to avoid the creation of isolated sets, if the new SELFLET would fill the Neighborhood of all the other SELFLETS as well as its own, the number of neighbors assigned to the new SELFLET will be equal to the maximum one imposed by the user minus one. This situation is depicted in Figure 3.1, where the maximum number of neighbors per SELFLET is fixed to 4. Once the SELFLET belongs to a Neighborhood, it can exchange messages with the neighbors, sharing information about its internal state. These information comprehend the utilization of the SELFLET, as well as the utilization, throughput and response time of services offered, and all the variables monitored by the SELFLET itself to define its internal state. A SELFLET is not aware of other SELFLETS that are not part of its Neighborhood. When a SELFLET leaves the system, it is removed from all the Neighborhoods.

Each SELFLET implements an adaptation loop, which allows the SELF-
Chapter 3. The SELFLET Framework

Figure 3.1: Graphical representation of neighbors management. The values in each circle represent the number of neighbors that could be added to the SELFLET.

SELFET to actuate an adaptation action on the basis of its internal state and the neighbors state. The behavior of the whole system emerges from the local decisions taken by each SELFET, exploiting the concept of emergent behavior.

Figure 3.2 represents an example of a SELFET based system. The application is composed of several services (Service1, Service2, ..., ServiceN) defined by the user through the framework. The system is composed of five nodes: one node running the Requests Dispatcher, i.e., the entry-point of the application, and the Message Broker used for the communication; four nodes where are deployed the SELFETs. Every SELFET provides different services that can be learned from other SELFETs.

The SELFETs are organized in three different Neighborhoods: this means that SELFETs \( n_1 \) and \( n_2 \) that belong to Neighborhood1 can communicate between them, as well as SELFETs \( n_3 \) and \( n_4 \) (Neighborhood2), and SELFETs \( n_2 \) and \( n_3 \) (Neighborhood3). As a consequence, SELFET \( n_1 \) is not aware of SELFET \( n_4 \) and cannot communicate with it.

When a client sends a request to the system, it is redirected by the Requests Dispatcher to a SELFET that provides that specific service. The request is computed, and if necessary it is forwarded to the next service until it is completed and the application can respond to the client. The adaptation loop inside every SELFET triggers adaptation actions that guarantee
that the system is always up and running and able to effectively satisfy the requests coming to the system.

3.2 Architecture

In this section we describe the core foundations of the SELFLET Framework focusing on a single SELFLET. The SELFLET conceptual model is depicted in Figure 3.2. Its main elements are services, that are high-level abstraction of the functionality provided by a SELFLET, the behaviors, that are the concrete implementation of the functionality, and autonomic policies, that trigger the autonomic actions and provide self-adaptive capabilities. A detailed description of these components is provided below.

3.2.1 Services

A service represents a high level task that can be executed by the SELFLET. A service describes only the functionality that can be achieved but does not specify its implementation, which is left to behaviors as described later. The SELFLET acts as a service container executing service requests: it can provide any number of distinct services to other SELFLETS. A service can be requested by a SELFLET (including the SELFLET that is currently offering it) and external components knowing the SELFLET’s endpoint.

Figure 3.2: Overview of a SELFLET based system.
Chapter 3. The SELFLET Framework

Figure 3.3: SELFLET conceptual model [42].

Services are formally defined by the Service Level Agreement (SLA). Despite it is possible to define several SLA for the services, in the current implementation we impose a constraint on the average Maximum Response Time ($R_{max}$) of the service, i.e., the maximum delay between the time instant when a request is received and the time when the result is obtained. Maximum Response Time is measured in milliseconds and the average is computed with a time windows ($\text{time}_w$) of 5 minutes.

3.2.2 Behaviors

The actual implementation of the functionality described by a service is maintained in a separate concept, that we call behavior. In this way, services can choose among different implementations, allowing the developers to implement different levels of quality and resource consumption for the same service. However, only one behavior is fixed for any given time instant (i.e., a service cannot be implemented at the same time by two different behaviors in the same SELFLET).

Behaviors are specified using simple state diagrams composed by states and condition on arcs. The states of a behavior correspond to the execution of one or more computations needed to complete a service, so a behavior represents the workflow needed to accomplish a specific service. We define
two types of behaviors: complex behavior and elementary behaviors.

Complex behaviors can be composed by any number of states without interconnection restrictions. Similarly to what happens to web service workflows, the semantic of each state is defined as an automatic request for the service indicated by that state. The services activated by states in a complex behavior can be implemented by complex behaviors themselves, allowing the creation of very elaborate workflows. A workflow is so composed by several complex behaviors that are executed sequentially and that can create different branches according to the values of defined parameters (e.g., the choices of the user on the tasks to execute).

Elementary behaviors are defined by a single state that represents a self-contained computation. This state cannot trigger a further service execution. This limitation avoid an infinite recursion in the invocation of complex behaviors: in order to execute the computation needed by a service, a complex behavior eventually triggers an elementary behavior. The code to execute the computation is contained in a software module called ability. As a consequence, every elementary behavior has an ability associated with it and can be directly executed by the SELFLET.

3.2.3 Autonomic policies

The SELFLET’s self-adaptive system is implemented by means of autonomic policies. Autonomic policies enable the adaptation of the SELFLET according to the internal state of the SELFLET and the information gathered from neighboring SELFLETS.

Autonomic policies can be of two different types: reactive policies and proactive policies. Although both types of policies allow the SELFLET to actuate an adaptation action, the action selection strategy and the policy activation time differ substantially. Given the decentralized structure of the system, each SELFLET evaluates policies independently from the others, producing an adaptation action to actuate. While adaptation actions provide the mechanism to perform a change on the current SELFLET, policies determine the kind and the actuation time for the adaptation.

Reactive Policies

The first type of adaptation policy is called reactive policy. Reactive policies are defined as ECA (Event-Condition-Action) rules [53], structured in a LHS (Left Hand Side) and a RHS (Right Hand Side). The LHS specifies an event – taken from the ones that are internally generated by the SELFLET – and a condition on that event. The RHS specifies the adaptation action
Chapter 3. The SELFLET Framework

that is actuated only if the condition on the LHS is satisfied (i.e., it reacts to an event).

Reactive policies provide a way for users to write their own policies according to their needs. Typically these policies are used for defining reactions to abnormal situations. For example, a user might specify to change a service implementation if a critical error happens during its execution.

Reactive policies have been implemented for completeness in the system, since they are the standard policies in cloud applications orchestration [22]. However, they are not part of our evaluation, which is focused on proactive policies.

Proactive Policies

Differently from reactive policies, proactive policies are triggered periodically – independently from any internal event. A proactive policy does not uses ECA rules. Instead, it is structured according to an adaptation loop which is in charge of generating the adaptation action to actuate. The adaptation loop is described in details in section 3.3.

Adaptation Actions

An adaptation action changes some configuration aspects of the SELFLET that actuates it. Action may have different effects on the SELFLET and are typically actuated in order to improve some aspects such as quality of service (e.g., response time), reducing costs (e.g., number of nodes instantiated), or reacting to abnormal conditions (e.g., recovering from a malfunctioning service). As a matter of fact, actions represent the possible adaptation mechanism invoked by the autonomic policies. Here we describe each of the five adaptation actions as well as its effects on the SELFLET and its neighbors.

- **Service Request Redirect.** Service request redirect is used to balance the load among SELFLETS. This actions involves a SELFLET \( n_1 \) actively redirecting service requests, and a SELFLET \( n_2 \) receiving them. In this way, \( n_2 \) processes a subset of the requests received by \( n_1 \). The choice of which requests to forward as opposed of being executed locally is done using a pre-request probabilistic approach.

  Service redirect action is defined with the tuple \( \langle s, n_1, n_2, p \rangle \), where \( s \) is the name of the service being redirected, \( n_1 \) is the SELFLET redirecting request, \( n_2 \) is the SELFLET receiving redirected requests and \( p \) is the probability that a single request is forwarded to \( n \). Service redi-
3.2. Architecture

rects are performed only toward SELFLETS that offer the requested service.

- **Service Teach.** A service teach action allows to scale a single service copying it from a SELFLET $n_1$ to a SELFLET $n_2$. The service is replicated completely, i.e., all the information about the name, input parameters, etc., as well as the behavior currently used by $n_1$ are copied in $n_2$. We can represent a service teach action with the tuple $\langle s, n_1, n_2 \rangle$ where $s$ is the service being taught by $n_1$ to $n_2$. The result of the execution of the action is that, from that point on, SELFLET $n_2$ will be able to execute the service locally.

- **Change Service Implementation.** This action can change the current behavior of a service with another one that offers the same functionality. This can be useful when two behaviors offer the same service but with different quality. For example, a video decoding service might have two codecs which differentiate each other by the amount of CPU time required to manipulate the video, and the quality of the produced video. Whenever the SELFLET is not under heavy load, it can execute the high quality codec, since the higher load generated by the codec will not likely saturate the SELFLET. Conversely, if the SELFLET is under heavy load, it can switch the service implementation with the codec using less CPU resources. A “Change service implementation” action is represented by the tuple $\langle s, b_1, b_2, n \rangle$ where behavior $b_1$ is replaced by behavior $b_2$ in service $n$ at SELFLET $n$.

- **Add SELFLET.** This action and the next one involve the scaling of the active SELFLETS in the system. This action is executed when a SELFLET $n$ and all its neighbors are under heavy load. The result is the creation of a new SELFLET that joins the system. Using this action SELFLETS can reduce the load that are facing, exploiting other actions, such as the teach and redirect one. The tuple representing an add SELFLET action is represented only by the id of the SELFLET requiring it: $\langle n \rangle$.

- **Remove SELFLET.** The effect of this action is to remove a given SELFLET, so it is the dual of the add SELFLET action. It is executed when the SELFLET current CPU utilization falls below a certain threshold. This action is used to avoid the waste of resources. The tuple identifying the action contains only the SELFLET that is requesting the action: $\langle n \rangle$.  

35
Chapter 3. The SELFLET Framework

![Diagram of SELFLET Framework](image)

Figure 3.4: Adaptation loop used by the SELFLET.

### 3.3 Adaptation Loop

The adaptation loop depicted in Figure 3.4 is periodically executed by each SELFLET to trigger autonomic policies and generate the autonomic actions to actuate. The loop is composed of two steps: the action generation and the action selection. During the action generation, all the possible action instances are generated according to the information received from neighbors and the internal state of the SELFLET. Action selection uses a probabilistic approach to select the best action to actuate. After the selection phase, the selected action is actuated.

Here, we describe the mechanism used to generate and select the autonomic actions. Since the actuation phase is trivial, it has no particular interest for the adaptation framework itself.

#### 3.3.1 Action Generation

We present the algorithm used for the generation of the action instances for each type of action implemented in the framework. The implementation of the algorithms related to the Teach and Change Implementation action is part of contribution C2, as well as the design of the Change Implementation algorithm. Table 3.1 summarizes the parameters used in the description of
3.3. Adaptation Loop

the algorithms. The internal state of the SELFET, as well as the one of its neighbors, is analyzed by each algorithm to generate a set of candidate adaptation actions. For each generated action is computed a weight between 0 and 1 (the higher the value, the higher the selection probability), that will be used by the selection algorithm. The candidate adaptation actions may vary over time, and depend on the state of the SELFET and its neighbors, e.g., when a SELFET “learns” a new service, the other ones will be able to redirect requests to that SELFET.

Table 3.1: System parameters.

| $\mathcal{N}$ | Set of SELFETs |
| $n$ | SELFET index |
| $\mathcal{S}$ | Set of services |
| $s$ | Service index |
| $B$ | Set of all behaviors |
| $B_s$ | Set of behaviors implementing service $s$ |
| $\Lambda_s$ | Direct requests for service $s$ |
| $R_s$ | Maximum average response time for service $s$ |
| $U_n$ | Utilization for SELFET $n$ |

The values $U_{max}$ and $U_{min}$ denote SELFET’s maximum and minimum allowed utilization threshold respectively. The maximum allowed utilization threshold is computed dynamically for each SELFET according to the equation:

$$U_{max} = \min(1 - \frac{D_k}{R_{max,k}})$$  \hspace{1cm} (3.1)

where $D_k$ is the expected demand of service $k$ and $R_{max,k}$ is the maximum response time defined for service $k$ (C1).

- **Redirect.** Redirect actions are generated according to Algorithm [1] For every locally available service, the algorithm generates a redirect action toward the neighbors of the SELFET offering it. The algorithm takes into consideration only the neighbors that are not exceeding the utilization threshold. The weight of a redirect action is computed by considering the utilization of the service $U_{n}$. Since the algorithms depends on the number of services and neighbors of the SELFET, its time complexity is $O(|\mathcal{N}| \cdot |\mathcal{S}|)$.

- **Teach.** Algorithm [2] used for the generation of the teach action, is similar to the one used for the redirect action. For every locally available
Chapter 3. The SELFLET Framework

Algorithm 1 Generation of adaption actions for service request redirect
1: redirectActions ← ∅
2: for \( s \in S_n \) do
3: \( \Delta U = \max(U_n - U_{\text{max}}, 0) \)
4: \( \Delta \Lambda_s^n = \frac{\Delta U}{\Lambda_s^n} \)
5: \( p = \min(\frac{\Delta \Lambda_s^n}{\Lambda_s^n}, 1) \)
6: for \( m \in N_n \) do
7: if \( s \in S_m \) and \( U_m < U_{\text{max}} \) then
8: action ← createRedirectAction(\( s, m, p \))
9: action.weight ← \( U^n_s \)
10: redirectActions ← redirectActions \( \cup \) action
11: end if
12: end for
13: end for
14: return redirectActions

Algorithm 2 Generation of adaption actions for service teach
1: teachActions ← ∅
2: for \( s \in S_n \) do
3: for \( m \in N_n \) do
4: if \( s \notin S_m \) and \( U_m < U_{\text{max}} \) then
5: action ← createTeachAction(\( s, m \))
6: action.weight ← \( U^n_s \)
7: candidateActions ← candidateActions \( \cup \) action
8: end if
9: end for
10: end for
11: return teachActions

service, the algorithm generates a teach action toward the neighbors of the SELFLET not offering it. The weight of the teach action depends on the current utilization of the service, so that the most used services will have higher probability to be replicated. The complexity is \( O(|N| \cdot |S|) \).

- **Change Implementation.** The algorithm generating the change implementation action for services (see Algorithm 3), can be divided in 2 cases: (i) the SELFLET and its neighbors are overloaded, so the system should change to a lower lever behavior – i.e., a behavior that requires less resources but provides a lower quality output – to improve performance, or (ii) the SELFLET and its neighbors are not overloaded, so the system can use a higher level behavior – i.e., a behavior that provides higher quality output but requires more resources – to im-
3.3. Adaptation Loop

Algorithm 3: Generation of adaption actions to change service behavior

1: changeActions ← ∅
2: $U_{\text{cur}} \leftarrow \text{averageSelfletUtil}()$
3: $U_{\text{neigh}} \leftarrow \text{averageNeighborUtil}()$
4: $S_{\text{over}} \leftarrow \text{overloadedServices}()$
5: if $U_{\text{cur}} > U_{\text{max}}$ and $U_{\text{neigh}} > U_{\text{max}}$ then
6:   for $s \in S_{\text{over}}$ do
7:     if $\exists b' \in B_s \setminus b$ s.t. $b$ is the current behavior of $s$ and $b' < b$ then
8:       action.weight $\leftarrow \max(\min(R_s - \frac{P_s}{P_s}, 0), 0)$
9:       action $\leftarrow \text{createChangeBehaviorAction}(s, b')$
10:      changeActions $\leftarrow$ changeActions $\cup$ action
11:   end if
12: end for
13: end if
14: if $U_{\text{cur}} < U_{\text{max}}$ and $U_{\text{neigh}} < U_{\text{max}}$ then
15:   for $s \in S \setminus S_{\text{over}}$ do
16:     if $\exists b' \in B_s \setminus b$ s.t. $b$ is the current behavior of $s$ and $b' > b$ then
17:       action.weight $\leftarrow \max(\frac{P_s - R_s}{P_s}, 0)$
18:       action $\leftarrow \text{createChangeBehaviorAction}(s, b')$
19:      changeActions $\leftarrow$ redirectActions $\cup$ action
20:   end if
21: end for
22: end if
23: return changeActions

prove the quality of the service. In case (i), if there exists a lower level behavior for the overloaded services (i.e., the ones that are violating the maximum response time), then an action is created with a weight that is proportional to how much the service is exceeding its maximum response time. Otherwise, in case (ii), if there exists an higher level behavior for the services that are not overloaded, then an action is created with a weight that is proportional to how much the response time of the service is lower than its maximum response time. The algorithm time complexity for generating all change implementation adaptation actions is $O(|S| \cdot |B|)$.

- Add SELFLET. When the level of the workload is higher than the capacity of the system, a new SELFLET is required to increment the available computational resources. Starting from this consideration, the add SELFLET action is generated whenever the average utilization of the SELFLET and its neighbors exceeds the utilization threshold $U_{\text{max}}$, as described in Algorithm 4. The algorithm relies on a partial
Chapter 3. The SELFLET Framework

Algorithm 4: Generation of adaption actions to add a new SELFLET

1: \( U_{\text{neigh}} \leftarrow \text{averageNeighborUtil}() \)
2: if \( U_{\text{neigh}} > U_{\text{max}} \) then
3: \( \text{action} \leftarrow \text{createAddAction()} \)
4: \( \text{action}.weight \leftarrow U_{\text{neigh}} - U_{\text{max}} \)
5: return \{\text{action}\}
6: end if
7: return \( \emptyset \)

Algorithm 5: Generation of adaption actions to remove a SELFLET

1: \( U_{\text{neigh}} \leftarrow \text{averageNeighborUtil}() \)
2: if \( U_{\text{neigh}} < U_{\text{min}} \) and \( |N_n| > 0 \) then
3: \( \text{action} \leftarrow \text{createRemoveAction()} \)
4: \( \text{action}.weight \leftarrow U_{\text{min}} - U_{\text{neigh}} \)
5: return \{\text{action}\}
6: end if
7: return \( \emptyset \)

view of the state of the system, which is derived from the information coming from neighbors. The idea behind this algorithm is inspired by a previous work [43]. The weight of the adaption action is computed as the difference between the average utilization and the utilization threshold. The time complexity of the algorithm is bounded by a constant term, i.e., the number of neighbors of the SELFLET set by the user in the configuration of the system.

- **Remove SELFLET.** The remove action follows the same rationale of the previously described action, being its dual. Algorithm 5 is used to generate remove actions. A SELFLET can be removed whenever its average utilization falls below the minimum utilization threshold \( U_{\text{min}} \). In order to keep at least one SELFLET in the system, the algorithm checks also that the Neighborhood of the current SELFLET is not empty. The time complexity of the algorithm is bound by a constant, like the one of the add SELFLET action.

### 3.3.2 Action Selection

The objective of the action selection phase is to choose a single action among the set of candidate ones produced in the action generation phase. However, choosing always the best action, i.e., the one with the highest weight, could bring the system in an undesired state. For example, if all the requests are redirected to the most unloaded SELFLET, that one will
3.4. Internal architecture

Algorithm 6 Algorithm for action selection

1: \( \text{totalWeight} \leftarrow \sum_{a \in \text{actions}} a.\text{weight} \)
2: \( \text{threshold} \leftarrow \text{rand}(0, 1) \)
3: \( \text{probability} \leftarrow 0 \)
4: \( \text{for } p \in \text{actions} \text{ do} \)
5: \( \text{probability} \leftarrow \text{probability} + \frac{a.\text{weight}}{\text{totalWeight}} \)
6: \( \text{if probability} > \text{threshold} \text{ then} \)
7: \( \text{return } a \)
8: \( \text{end if} \)
9: \( \text{end for} \)

become quickly overloaded. This behavior could affect also the resources of the system, leading to resources waste. For example, if too many SELF-LETS join the system following the actuation of “add SELFLET” actions, the resulting system will be highly over-provisioned. To overcome this situation, we use a weighted random selection, i.e., the action is randomly selected according to a probability proportional to its action weight. This approach is based on Algorithm 6.

Once the \( \text{totalWeight} \) of the created \( \text{actions} \) is computed, a \( \text{threshold} \) is randomly chosen between zero and one. The list of \( \text{actions} \) is then traversed, adding the normalized weight of the current action to the value of a \( \text{probability} \) variable (previously initialized to zero), and checking if its value is greater than the \( \text{threshold} \). When the value of \( \text{probability} \) is greater than the threshold, the current action is selected by the algorithm.

Using the proposed approach, it is possible to select actions that have a worse \textit{estimated} result. This behavior is beneficial to exit potential local optima, and can be found in various works [43,58,110,147]. The principle behind it is often called \textit{noise} [77,110].

3.4 Internal architecture

In this section we provide some details regarding the internal architecture and the implementation of the SELFLET framework. The architecture is shown in Figure 3.5.

Dispatcher

The interaction between the internal components of the SELFLET is done using an event driven paradigm through a central event dispatcher. The dispatcher offers an interface to components for the publication of new events, and delivers events to components subscribed to a given one.
Chapter 3. The SELFLET Framework

Figure 3.5: The internal architecture of a SELFLET.

Message Handler
The message handler has to translate network messages into internal events and vice-versa. It uses the underlying network infrastructure to send and receive messages.

Communication Manager
The functionalities of the Communication Manager can be grouped in 3 categories:

- interpreting and handling events produced or received by the message handler. In particular, for each event associated to a message, the communication manager defines a specific message handler performing the appropriate action (for example, given a service request event, a handler starts a new service instance);

- management of aspects related to the communication with other SELFLETS such as performing service teach, service requests, etc.

- managing the neighboring relationships. In particular, a SELFLET keeps a list of SELFLET neighbors which can be directly contacted
3.5. Conclusion

by the communication manager (i.e., SELFLETS which are “known” by a given SELFLET).

Service Executor

The Service Executor manages the actual execution of services by allocating them to threads. The service executor maintains a configurable thread pool and a queue of tasks to be executed.

Autonomic Manager

The Autonomic Manager implements the self-adaptive core of the SELFLET architecture, and is responsible for the execution of the autonomic policies. It implements the “execute” phase of the MAPE loop by providing a unified interface grouping all operations that can be invoked by autonomic policies. Example of these operations are: changing service implementation, teaching a service, redirecting service requests, etc.

Monitoring Manager

The Monitoring Manager provides the functionalities for monitoring the execution of internal services and offers a unified interface to access this information.

Internal Knowledge

The Internal Knowledge stores data during the SELFLET life-cycle and is structured in four parts: (i) Knowledge Base, used to store and retrieve generic information needed by any of the SELFLET components; (ii) Service Repository, which lists the services offered by the SELFLET; (iii) Behavior Repository, which contains all the behaviors specifications the SELFLET is able to run; (iv) Attribute Repository, which stores the description about the SELFLET.

3.5 Conclusion

In this chapter we presented the SELFLET Framework, a decentralized self-adaptive framework to build distributed service-based applications. We described the architecture of the system, presenting its main components such as services, behaviors and policies, as well as the adaptation loop implemented inside every SELFLET for the decision making process. The autonomic actions that the system can actuate involve both the management of resources (Add or Remove SELFLET) as well as the behavior of
Chapter 3. The SELFLET Framework

the SELFLET itself or the services it provides (*Redirect, Teach, Change Implementation*).

These concepts are the core of the SELFLET Framework, and are needed to understand its evaluation that is presented in the next chapter. There, we describe how we designed and deployed in a cloud infrastructure a realistic use case built with the SELFLET Framework.
CHAPTER 4

Evaluation of the SELFLET Framework

In the previous chapter we presented the SELFLET Framework and described its architecture and features. The framework has been already evaluated in a previous work, using simulations and small problems [42]. However, we want to evaluate the SELFLET in a real environment to provide a strong validation of the concepts behind it, and to better understand its strengths as well as its weaknesses that may prevent its adoption in a production environment. To reach this goal, we designed a realistic case study and implemented it using the SELFLET Framework itself. We deployed our application in a cloud environment composed of several Virtual Machines (a maximum of 55 VMs) and tested it using two different workloads. The first type of workload has been used to test the capability of the system to adapt to quick changes in the traffic coming to the application, while the second one has been obtained analyzing the traffic coming to a real website and has been used to test if the system is able to handle a realistic workload. This chapter includes the contributions C3, C4 and C5 to the SELFLET Framework defined at the beginning of Chapter 3.

The chapter is organized as follows: Section 4.1 presents the case study we developed (C4); it is derived from a video-on-demand scenario, a current topic that perfectly fits the type of applications that could be developed
Chapter 4. Evaluation of the SELFLET Framework

using the SELFLET Framework. After the case study, we describe in details the evaluation process in Section 4.2; first we present the objectives of the experiments, then we describe the setup of the system and show the obtained results (C3, C5). The discussion of the results is in Section 4.3 while Section 4.4 concludes the chapter.

4.1 Case study

Video-on-demand applications account for 70% of the internet traffic share in 2015, with Netflix that accounts for more than a third of internet traffic, and Cisco predicts that by 2020 82% of the world’s internet traffic will be video [52]. Starting from this consideration, we developed a case study that simulates a video-on-demand cloud application. Clients request a video and can operate modifications on it (e.g., scaling the video) and add subtitles that will be embedded inside the video itself.

In our case study we assume that clients send requests for video resources including the requested media as well as the required video format. The provider is in charge of retrieving the video and then processing it using its computational infrastructure which we assume to be based on the SELFLET Framework. The major task of the provider is to adapt the videos according to the user preferences such as screen resolution, bandwidth, language, etc. Beside the actual satisfaction of its users, the provider is also interested in minimizing the consumed resources – thus its costs – for example by avoiding the over-allocation of SELFLETS. We also assume that all video files as well as other resources (i.e., subtitles) are stored in a network file system or a web storage service such as Amazon S3 [1] which can be easily accessed by all SELFLETS.

The problem fits well the characteristics provided by the SELFLET for the following reasons:

- The video processing task faced by the provider can be easily structured in well-defined steps – services and behaviors – which might be spread over a network of SELFLETS.

- Some video processing services can have multiple implementations using different codecs such as [10, 21].

- Video processing is a heavy-computational task which can require a high number of nodes (i.e., large size of the system).

- Since workload is generated by users in an unpredictable manner, a scalable software framework is needed in order to cope with sudden
4.1. Case study

workload increase.

The application has been implemented simulating the usage of the CPU according to the different services. This is a simplification that has no impact on the evaluation of the self-adaptive capabilities of the decentralized system: the self-adaptive actions are mainly based on the load that is facing the systems, that can be effectively represented by CPU usage and the execution time of the services, since the simulated application is mainly based on a computational task. However, in the future we plan to apply our concepts to a complete application, implementing the different operations of the services instead of simulating their duration and CPU usage.

The same case study will be implemented as a microservices application and used in the evaluation of the GRU tool in Chapter 7.

Case study implementation – SELFLET

Here we describe the main services and behaviors of the SELFLET application for video decoding. The main service – invoked upon a client request – is depicted in Figure 4.1 and it is named videoProvisioner. This service defines the major activities performed in order to process the video.

Firstly, it retrieves the video resource as indicated by the client using the service getVideoResource. This service, depicted in Figure 4.2, first checks whether the service was already downloaded by the provider (to avoid a second download) and if not, retrieves it from the source indicated by the client.

After downloading the video, the service provider adapts the video resolution as required by the client using the scaleVideo service. Its implementation is shown in Figure 4.3 and is composed by two alternative elementary behaviors using two different codecs.

The next two states of the behavior implementing the videoProvisioner service deal with two more steps of video processing: transcoding (i.e., changing the encoding of the video to another format) and bit depth reduction (i.e., reducing the number of bits to represent a pixel). We assume these two services implemented with the corresponding abilities and for this reason we do not report their (trivial) behavior.

The final step that the service provider must deal with is the application of subtitles to the clip. This task is optional and depends on the client request. The task is divided in two services: (i) getSubtitles for retrieving the correct subtitles and, (ii) applySubtitles to actually applying the subtitles to the video resource. While the implementation of the applySubtitles service is defined with an ability, the implementation
Chapter 4. Evaluation of the SELFLET Framework

Figure 4.1: Workflow – behavior – for a `videoProvisioner` service. It is composed of several sub-services that are executed sequentially.

of the `getSubtitles` is slightly more complex. In particular, its implementing behavior is reported in Figure 4.4 and is structured in the following manner: firstly it checks whether the requested subtitles are available locally and, if so, the reference to the file containing the text is returned im-
4.1. Case study

![Diagram of getVideoResource service behavior]

Figure 4.2: Behavior of getVideoResource service. If the video resource is not present locally, it is downloaded from remote.

![Diagram of scaleVideo service behaviors]

Figure 4.3: Two alternative behaviors for service scaleVideo. Every implementation can provide the same service but with different quality.

mediately. The service checks whether the subtitles exist in the requested language, otherwise returning them in the positive case. When the subtitles
Chapter 4. Evaluation of the SELFLET Framework

Figure 4.4: Behavior of getSubtitles service. It is composed of several sub-services to manage the subtitles of the video resource.

are not available, the reference to the file containing subtitles in a default language is retrieved and then translated using an external translation service (for example [12]).

4.2 Evaluation

We tested the SELFLET Framework in the presented case study to validate it. The objectives of the experiment are the following:

- **OBJ1**: show that the system provides the requested services minimizing the violations of the maximum response time defined in the Service Level Agreement (SLA), and quantify the violations;
- **OBJ2**: show that the system self-adapts according to the workload
4.2. Evaluation

minimizing the waste of resources;

- **OBJ3**: show that all the autonomic actions we have defined are used to self-adapt

- **OBJ4**: compare a static and a dynamic threshold for the automatic scaling of the system (i.e., the capability of the system to autonomously add and remove components if needed).

We want to focus the evaluation on the performance of the system related to the adaptation of a complex application, while an evaluation of the effectiveness of the framework for the design of the application has been the focus of a previous research [42].

4.2.1 Setup

The maximum number of possible active SELFLETs is fixed to 55 due to a limitation in the maximum amount of available resources imposed by a budget constraint. Every instance runs a single SELFLET to avoid a possible overload of the VM. The workload has been studied to take into account this limitation.

Since Amazon Web Services charge the customer per active instance per hour, a SELFLET can be removed only in the last 15 minutes of each hour. The system can start a maximum of 3 SELFLETs every 5 minutes, with the Request Dispatcher that can discard scaling-out requests coming from the SELFLETs. This limit has been imposed to let the system stabilize before the allocation of new resources, and can be useful to avoid an excessive growth of the system in a situation where the available resources are limited. The maximum number of new instances, as well as the time frame, have been chosen through an empirical evaluation done with dedicated experiments.

The first instantiated SELFLET provides all the services, while each new SELFLET starts without any service: this choice allows to test the service teach autonomic action. The number of neighbors of each SELFLET has been limited to 5 to test the interaction among different neighborhoods. The starting number of SELFLETs has been fixed to 3 as a bootstrap of the system.

The expected demand of each service has been chosen between 500ms and 1500ms for simple services on the basis of the type of service, while for composed services it is the weighted sum of the expected demand of the subservices. The only exception is the demand of scaleVideo service, that has been obtained from the analysis of FFmpeg video scaling benchmark.
Chapter 4. Evaluation of the SELFLET Framework

in [119]. The demand of each service is generated according to a random exponential distribution with \( \lambda \) equal to the expected demand of service. The Maximum Response Time of services is a value chosen between 2 to 5 times the expected demand, according to the type and expected demand of service. The values related to the demand and maximum response time of the services have been chosen according to the functionality provided by the service, and are consistent with a realistic scenario [20].

In order to simulate a real interaction and the uncertainty of user behavior, the transition between one service and the next in the workflow of the application presents a defined probability to be executed, which has been set depending on the type of service. Table 4.1 shows the demand, the maximum response time and the probability to be executed for each service, as well as the computed scaling upper threshold (see equation 3.1 for details) and the violations occurred during the tests.

Table 4.1: Services parameters and violations.

<table>
<thead>
<tr>
<th>Service Name</th>
<th>( D_k )</th>
<th>( R_{\text{max},k} )</th>
<th>( \bar{C}_{\text{max}} )</th>
<th>( P )</th>
<th>Workload1 viol.</th>
<th>Workload2 viol.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dynamic Static</td>
<td>Dynamic Static</td>
</tr>
<tr>
<td>applySubtitles</td>
<td>600</td>
<td>1200</td>
<td>0.5</td>
<td>0.2</td>
<td>1.9% 9.3%</td>
<td>4.4% 5.0%</td>
</tr>
<tr>
<td>bitDepthReduction</td>
<td>1300</td>
<td>3900</td>
<td>0.6</td>
<td>1.0</td>
<td>0.0% 0.9%</td>
<td>0.0% 0.5%</td>
</tr>
<tr>
<td>checkIfAvailable</td>
<td>1200</td>
<td>2400</td>
<td>0.5</td>
<td>1.0</td>
<td>1.9% 4.2%</td>
<td>1.1% 1.9%</td>
</tr>
<tr>
<td>checkSubAvailable</td>
<td>700</td>
<td>2100</td>
<td>0.6</td>
<td>0.2</td>
<td>0.5% 0.5%</td>
<td>0.5% 1.1%</td>
</tr>
<tr>
<td>checkSubExists</td>
<td>1500</td>
<td>3000</td>
<td>0.5</td>
<td>0.95</td>
<td>5.1% 9.3%</td>
<td>5.3% 5.5%</td>
</tr>
<tr>
<td>downloadVideo</td>
<td>1500</td>
<td>4500</td>
<td>0.6</td>
<td>0.95</td>
<td>0.0% 0.0%</td>
<td>0.3% 0.0%</td>
</tr>
<tr>
<td>getDefaultSubtitle</td>
<td>1200</td>
<td>2400</td>
<td>0.5</td>
<td>0.9</td>
<td>7.9% 5.8%</td>
<td>4.4% 5.3%</td>
</tr>
<tr>
<td>getSubtitles</td>
<td>3279</td>
<td>13117</td>
<td>0.75</td>
<td>0.2</td>
<td>0.0% 0.0%</td>
<td>0.0% 0.0%</td>
</tr>
<tr>
<td>getVideoResource</td>
<td>2625</td>
<td>13125</td>
<td>0.8</td>
<td>1.0</td>
<td>0.0% 0.0%</td>
<td>0.0% 0.0%</td>
</tr>
<tr>
<td>scaleVideo</td>
<td>28500</td>
<td>85500</td>
<td>0.6</td>
<td>1.0</td>
<td>0.0% 0.0%</td>
<td>0.0% 0.0%</td>
</tr>
<tr>
<td>transcodeVideo</td>
<td>900</td>
<td>1800</td>
<td>0.5</td>
<td>1.0</td>
<td>2.8% 4.2%</td>
<td>1.4% 1.4%</td>
</tr>
<tr>
<td>translateSub</td>
<td>500</td>
<td>1500</td>
<td>0.6</td>
<td>0.3</td>
<td>3.3% 3.3%</td>
<td>1.8% 3.2%</td>
</tr>
<tr>
<td>videoProvisioner</td>
<td>34100</td>
<td>102302</td>
<td>0.6</td>
<td>1.0</td>
<td>0.0% 0.0%</td>
<td>0.0% 0.0%</td>
</tr>
</tbody>
</table>

Scaling threshold setup

We test two implementations of the scaling strategy for the SELFLETS: (i) a static threshold with a scaling upper threshold fixed to 60% CPU usage; (ii) a dynamic threshold that computes the scaling upper threshold according to equation [3.1]. For both the implementations the scaling lower threshold has been set to 40% CPU usage. The values of the thresholds are the results of
4.2. Evaluation

a fine-tuning phase done with dedicated experiments.

Hardware setup

In order to achieve our objectives, we deployed the system on the Amazon Web Service infrastructure, using the EC2 Amazon m3.large instances (64 bit Processor Architecture with 2 vCPU, 7.5GiB of Memory and 1 x 32GB SSD Storage). We limited the number of active SELFETs on each VM to one, to avoid the overload of the computing instance. The EC2 machines were preallocated, while each time a new SELFET instance is required the code is automatically downloaded from GitHub and compiled using Apache Maven. The preallocation of machines allowed us to use spot instances and reduce the costs of the experimental evaluation.

Workload setup

We generate the workload for the test using Apache Jmeter [3]. We use this tool to send HTTP requests for service videoProvisioner – i.e., the main service – to the Requests Dispatcher that is the entry point of the system. Jmeter has been deployed on a dedicated Amazon instance in the same region of the instances where the SELFETs are deployed. We defined two traffic shapes which are shown in Figure 4.5.

The first one holds for 3 and a half hours and presents a step between 0.4 and 0.8 RPS to test the capability of the system to adapt to quick changes in the load. The second one holds for 6 hours and presents a more realistic bimodal traffic shape based on real data: the traffic of 48 hours of a real website has been compressed in 6 hours and scaled between 0.1 and 0.8 RPS. The traffic has been compressed in order to reduce the execution time of the experiment and to evaluate the responsiveness of the system. The website monitored belongs to a car manufacturer and the data are not publicly available. Despite the domain is different from the one of the case study application, the extracted workload reflects the traffic that a public website of an international company can face.

Baseline setup

As a baseline we simulated a traditional static system that is not able to actuate any autonomic action. The system is composed of a fixed number of SELFETs that provide all the services. The system cannot scale and the number of active SELFETs is fixed to 45 instances. The number of SELFETs has been chosen according to the equation:
Chapter 4. Evaluation of the SELFLET Framework

Figure 4.5: Workload used for the evaluation. Workload1 is used to test how the system react to changes in the traffic, while Workload2 represents a realistic traffic shape.
4.2. Evaluation

\[ N_s = \frac{\lambda \cdot D}{U_{\text{max}}} \]  

(4.1)

Where \( N_s \) is the number of active SELFLETS, \( \lambda \) are the RPS for service videoProvisioner, \( D \) is the expected demand of the service and \( U_{\text{max}} \) is the maximum CPU utilization. \( N_s \) has been chosen considering \( \lambda = 0.8 \) (the peak of the workloads) and \( U_{\text{max}} = 0.6 \).

4.2.2 Experimental results

We evaluate the SELFLET framework with two experiments using the workloads described in the previous section. In both the tests we compare the dynamic and static scaling threshold between them and against the baseline system (OBJ4). The figures in this section represent the average of the results of all the runs with a time mean of 5 minutes.

In the first test we address the self-adaptation capability of the system and the speed of the scaling algorithm. We used workload1 that is depicted in Figure 4.5a. In the second test we address the capability of the system to manage a realistic workload using workload2 that is depicted in Figure 4.5b. We collected the data of 10 runs, 5 using the static scaling algorithm and 5 using the dynamic one as well one run of the baseline system.

Response time

Figure 4.6 shows the results of the experiment with respect to the response time of service videoProvisioner. Both the dynamic and scaling thresholds are able to keep the response time of the requests around the one provided by the baseline system, with the dynamic scaling threshold performing slightly better than the static one (OBJ1).

Scaling

The scaling of the system is depicted in Figure 4.7 where the chart shows the total number of active SELFLETS per time step using the two different scaling thresholds (dynamic and static). The system is able to scale following the traffic shape (OBJ2), with a peak of 52 active SELFLETS using the dynamic strategy with workload1 (Figure 4.5a).

Utilization

The average utilization of the SELFLETS is represented in Figure 4.8. The chart shows the capability of the system to balance the load once the system
Chapter 4. Evaluation of the SELFLET Framework

![Diagram of videoProvisioner response times with Workload 1 and Workload 2]

Figure 4.6: Response times of service videoProvisioner. The response time of the application is always under the threshold defined in the SLA, with both the workloads.
4.2. Evaluation

Figure 4.7: Active SELFLETS. The number of active SELFLETS follows the traffic shape, with the use of the dynamic strategy leading to a greater number of instances.
Chapter 4. Evaluation of the SELFLET Framework

has been stabilized, keeping it around the threshold imposed for scaling (OBJ2). This is mostly evident with workload1, where the load is kept constant for some time before and after the step (Figure 4.8a). The baseline system is clearly underused, with a waste of resources which translates in higher costs.

![CPU utilization average](image)

(a) Workload1

![CPU utilization average](image)

(b) Workload2

**Figure 4.8:** SELFLETS utilization over time. While the baseline wastes resources when the traffic is low, the SELFLETS can scale the instances to keep the usage of resources around the 60%.
4.3. Discussion

Autonomic actions

The charts in Figure 4.9 represent the percentage of actions actuated by the SELFLETS among the possible ones. The comparison is between the static scaling threshold and the dynamic scaling threshold in the test with workload2 (Figure 4.5b). The actions are logged after the decision making process of each SELFLET. In this way, the constraint imposed on the maximum number of SELFLETS that can be started in 5 minutes (see Section 4.2.1) does not affect the validity of collected data. The majority of actions are not related to scaling, demonstrating the importance of the other autonomic actions which are usually not present in other self-adaptive systems (OBJ3).

![Figure 4.9: SELFLETS actions. The most actuated actions are not related with scaling, proving the value of different autonomic actions in the self-adaptive process.](image)

4.3 Discussion

According to the presented results the SELFLET Framework is capable to effective scale autonomously to guarantee a response time which does not violate the defined SLA. The system can scale accordingly to the variable traffic workload, adding and removing SELFLETS in order to satisfy the current demand and avoiding under- and over-provisioning of resources, as shown in Figure 4.7.

Table 4.1 compares the results obtained using the dynamic scaling threshold with the static one. Using the dynamic scaling threshold, the system is able to serve the majority of the service requests without violate the maximum response time. The 100% of the requests of the main service – i.e.,
Chapter 4. Evaluation of the SELFLET Framework

videoProvisioner service – are satisfied with a response time which respects the SLA. The dynamic threshold can guarantee less violations for a greater number of requests for almost all services compared to the static one, with an improvement up to 7.4% with respect to service applySubtitles with workload1.

The drawback of the use of a dynamic threshold is a higher number of active SELFLETS compared to static scaling threshold. This translates in higher costs for the users, so the tradeoff between performance and cost should be taken into account. However, the capability of the dynamic scaling threshold to self-tune can allow the user to avoid the tuning phase to determine the best threshold for scaling.

The baseline system lead to a waste of resources (and money) when the workload is low (Fig. 4.8), while the SELFLET Framework can provide almost the same performance with an effective balancing of load among the available resources.

Looking at Figure 4.9 we can observe the importance of autonomic actions different from the ones related to adding or removing SELFLETS: the majority of the actuated actions are not related to scaling. These data demonstrate the ability of the system to self-adapt to the changing workload balancing the load in an autonomous way, avoiding to scale as a primary strategy of adaptation.

The presented evaluation has some limitations. The services implementation is based on a function that keeps the CPU busy for a time which depends on the hardcoded demand of the service. This cannot be compared to a real system where each service presents different performance and demand according to its implementation and the hardware of the machine on which it is running. However, the focus of our evaluation is the autonomic capability of the SELFLET framework, and the simplification of the services implementation has no significant impact on it.

The instances provided by Amazon are allocated in advance to allow the utilization of spot instances and reduce the costs. In order to take advantage from the dynamic scaling of the system the instances should start on demand.

Due to a limitation on the number of active Amazon instances imposed by a budget constraint, the maximum number of active SELFLETS is fixed. Despite this could be a relevant limitation in a real environment, we overcome this constraint with the study of a workload that was suited for the maximum number of SELFLETS we can start.

Overall, we can conclude that the results of the experiment validate the SELFLET Framework in the presented case study.
4.4 Conclusion

In this chapter we presented the evaluation we made on the SELFLET Framework. The application we used for our evaluation has been developed using the framework itself, and represents a complex and realistic case study inspired by the context of video-on-demand applications.

We described the setup of the system and the application that have been deployed on a cloud infrastructure composed of more than 50 nodes, providing the two different workloads used to test both the capability of the system to adapt to a sudden change in the workload and the capability to handle a realistic workload obtained monitoring the traffic of a real website.

The obtained results validate our approach as we highlighted in the discussion, where we point out also the limitations of our system.

The evaluation of the SELFLET Framework presented in this chapter is the base of its critical analysis that will be presented in the first part of the next chapter.
CHAPTER 5

Enabling Transparent Adaptation in Complex Systems

The results obtained in the evaluation of the SELFLET Framework in the previous chapter validate its approach for the creation and management of a decentralized self-adaptive system. In this chapter we start with the critical analysis of the design choices at the base of the SELFLET Framework, pointing out its advantages as well as its limitations.

This analysis helped in the design of a tool that can enable the transparent adaptation in complex systems, i.e., it actuates the self-adaptive process externally from the system, without requiring its modification. This tool is applied to an external distributed system in order to introduce in it self-adaptive capabilities in a transparent way. The tool inherits the advantages of the SELFLETs but tries to overcome their limitations. We propose the division of the distributed Self-Adaptive System in three distinct parts: the adaptation manager, the managed element, and the adaptation enabler. In this way we separate the adaptation logic – inside the adaptation manager – from the application we want to manage that represents our managed element. The adaptation enabler is the contact point between these two parts, and it is a virtual container that encapsulate the application and makes the
Chapter 5. Enabling Transparent Adaptation in Complex Systems

interaction between the adaptation manager and the managed element possible.

We do not stop at the design of such tool, but we reason about its technological implementation, identifying the Microservices Architecture and the Docker containers as the key technologies for its concrete deployment. The Microservices Architecture presents all the characteristics that are fundamental for the distributed system we want to self-manage – i.e. our managed element – so it represents a perfect application domain for our tool. Docker containers are the de facto standard adopted by industry as virtual containers and can be used as the implementation of the adaptation enabler. Despite Microservices Architecture and Docker are quite new technologies, they have been the topic of some researches concerning Self-Adaptation, that we describe to position our work in relation to the current state of the art.

This chapter is organized as follows: Section 5.1 discusses advantages and limitation of the SELFLET Framework. The result of this analysis is the design of the tool to enable transparent adaptation in complex systems that is presented in Section 5.2. The technological implementation of the tool is the topic of Section 5.3 where the Microservices Architecture and Docker containers are introduced. Section 5.4 provides an overview of the role of microservices and Docker in the research concerning Self-Adaptation. The last part of the chapter is the conclusion presented in Section 5.5.

5.1 Advantages and limitations of the SELFLET Framework

In Chapter 4 we provided an extensive evaluation of the SELFLET Framework, obtaining good results and validating the SELFLETs using a realistic case study.

Here, we want to analyze the design choices at the basis of the SELFLET Framework to understand both its advantages and limitations.

Distributed System

The SELFLETs are self-adaptive agents distributed among a large-scale infrastructure composed of several nodes, running a service-based distributed application. This choice makes the system highly scalable, but both the application and the self-adaptive process may be affected by the problems of a distributed system, that are usually related to the synchronization between the various components, unreliability of the network and consistency of the data. Regarding the distributed application, we assume that it is designed and developed to handle all the issues related to this type of systems. The
5.1. Advantages and limitations of the SELFLET Framework

problems related to the distribution of the self-adaptive process are solved through decentralization. SELFLETS act as total independent units, so there is no need for synchronization between them, and possible issues like the delay in the actuation of actions have not a strong impact on the whole system. For the same reason, the unreliability of the network and the communication channels does not affect significantly the self-adaptive process. The knowledge-base of every SELFLET is local and independent from the one of others, so it is not affected by consistency problems. However, the loss of information or consistency of data may affect the quality of the decision on the self-adaptive actions to actuate. This may result in the actuation of actions that are not the best ones for the system.

Decentralization

The use of a decentralized approach based on a Multi-Agent System proved to be a good choice. The system does not present a single-point of failure, and allows to scale with ease adding and removing components at runtime without affecting the overall performance of the application (Figure 4.7).

Possible issues of a decentralized approach are the communication overhead deriving from the communication between the agents and the coordination between the agents.

These are both solved using the self-organization in Neighborhoods and the concept of emergent behavior. Since the agents organize themselves in sub-sets of peers to communicate with, the communication overhead is reduced even in a system composed of thousands of nodes. This choice leads to the lack of a global knowledge of the state of the system, since an agent has only access to the piece of information that is part of its own subset. However, we proved that the use of partial knowledge can ensure an effective self-adaptation, as demonstrated in Figure 4.6 and Figure 4.9. The Figures show that the system can ensure that the response time of the application is always under the Maximum Response Time defined in the SLA, and that the SELFLETS can chose the most effective adaptation action to achieve such goal (i.e., avoiding to always choose the adaptation actions related to the scaling of the application).

Emergent behavior

Every agent decides autonomously the actions to actuate, and the self-adaptive capabilities of the system emerge from the local decisions of every agent (emergent behavior). The agents composing the system does not have to explicitly coordinate among them to decide the actions to actuate, reduc-
Chapter 5. Enabling Transparent Adaptation in Complex Systems

ing the complexity of the decision making process. This choice also helps in the reduction of the communication overhead, since the communication cost of the coordination between agents is zero.

Despite emergent behavior may lead to instability of the system, it has been used extensively in literature especially in bio-inspired approaches (see Chapter 2 for details about bio-inspired approaches in Self-Adaptation), and the use of a probabilistic algorithm – like Algorithm 6 presented in Chapter 3 – for the decision of the adaptation actions to actuate can reduce this instability.

Adaptation policies

The use of an adaptation loop based on the MAPE-K feedback loop and policies is a consolidated approach (as described in Chapter 2). The policies and actions that enable the adaptation process have the advantage to be easily customizable by the users. This is not a minor issue, since we want to design a system that can be adopted by practitioners, so they should have the possibility to personalize the system according to their needs.

Service container

The SELFLETS are basically containers that manage the services inside them. This provides several advantages, like the encapsulation of the application that can be managed in blocks, and the isolation with other services running in the host machine.

A single SELFLET can contain several services and can teach them to other SELFLETS. This is a choice that improves the adaptability of the system. However, the encapsulation of single services in a container may lead to a better use of resources, since it should be possible to allocate the correct amount of resources for each service. Moreover, if a SELFLET fails, all the services inside it will be unavailable and it may take some time to be recovered.

Decoupled application

The use of an application based on decoupled services allows a fine grained management of the application and the resources (see Figure 4.8). Moreover, it is easier to handle failures of the services: if a service fails, a SELFLET can easily remove and replace it with a new one, without compromising the entire application or the SELFLET itself.

The distinction between the service and its implementation (called Behavior in the SELFLET Framework) enable a more effective adaptation,
5.1. Advantages and limitations of the SELFLET Framework

since it is possible to adapt even a single component of the application.

Integration

The SELFLETs use an internal approach for the integration with the application: the services of the application are developed using the SELFLET Framework and run inside the SELFLETs. This choice has the advantage to provide a better control over the application.

However, the consequence is that an application that is already developed and deployed cannot become self-adaptive unless it is redesigned using the framework.

Framework

The usage of a framework for the development of a SELFLET-based application can help the developers in the design and implementation of the services composing it. Developers are guided through the design of the system and can focus on the functional part of the services they have to implement.

The drawback of the use of a framework is that developers need to learn how to use it. Moreover, this poses a serious constraint on the technology stack of the application, that is imposed by the framework itself. The ideal scenario is that every service is built using the technology or language the developers are more comfortable with, and that provides the best performance for the task the service has to accomplish.

Analysis summary

Table 5.1 summarizes our analysis about advantages and limitations of the SELFLET Framework. In general, the majority of the design choices have proven to be good, compensating for limitations between them (e.g., the decentralization and emergent behavior) or presenting possible issues that require just a small effort to be addressed (e.g., the design of policies).

As major issues, we identified the ones related to the utilization and integration of the framework itself. The use of a framework forces the developers to learn how to use it, and imposes some constraints on the technologies used for the development of the application. The internal approach makes impossible to easily integrate the adaptation logic inside a deployed system. Since we want to enable Self-Adaptation easily in any distributed system, these are major issues that need to be addressed.

Some improvements can be made also in the use of containers, so we need to study the best way to exploit this idea that has proven to bring great
Chapter 5. Enabling Transparent Adaptation in Complex Systems

advantages in terms of the management of the application.

Table 5.1: SELFLET Framework advantages and limitations.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Advantages</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>high scalability</td>
<td>loss of information could reduce the effectiveness of the decision making process</td>
</tr>
<tr>
<td>Decentralization</td>
<td>fault-tolerance; scalability</td>
<td>communication and coordination costs</td>
</tr>
<tr>
<td>Emergent behavior</td>
<td>reduced coordination; reduced communication overhead</td>
<td>possible instability of the system</td>
</tr>
<tr>
<td>Adaptation policies</td>
<td>easy to customize</td>
<td>require careful desing</td>
</tr>
<tr>
<td>Service container</td>
<td>encapsulation; isolation; service management</td>
<td>resource management; container failure</td>
</tr>
<tr>
<td>Decoupled application</td>
<td>fine grained management; fault tolerance; improved adaptation</td>
<td>distributed systems issues</td>
</tr>
<tr>
<td>Integration</td>
<td>better control over the application</td>
<td>integration with deployed systems</td>
</tr>
<tr>
<td>Framework</td>
<td>help in the design and development</td>
<td>need to learn the framework; technological constraints</td>
</tr>
</tbody>
</table>

5.2 A tool to enable transparent adaptation in complex systems

The identified limitations in the analyzed system are related to the integration of the adaptation logic inside the application itself, and in the difficulty for a user to learn and use a new framework. Moreover, this approach based on a framework makes impossible to adapt with ease a system that has already been deployed. Other improvements can be done in the encap-
5.3. Technological considerations

In the previous section we described a possible design for a tool to bring transparent Self-Adaptation to distributed systems that we derived from the
Chapter 5. Enabling Transparent Adaptation in Complex Systems

analysis of the SELFLET Framework in Section 5.1. Here we want to make some considerations about the technologies that can be used for the implementation of such tool. The concrete implementation of the tool is provided in the next chapter, where we present a prototype called GRU.

5.3.1 Adaptation manager

The adaptation manager is an external and independent component, built as a Multi-Agent System composed of adaptation agents, that is applied to the managed element through the adaptation enabler. This implies that it can be implemented using any technology and language, because it has no impact on the managed element. The important aspect to consider is its customization, because we want to make it applicable to a wide range of possible applications. The customization of the adaptation manager can be done using external files written in a specific language (e.g., JSON, XML, YAML, etc.) that are loaded when agents start, or using an external distributed repository where the configuration is stored and shared between agents.

5.3.2 Managed element: Microservices Architecture

The managed element is an application composed by decoupled and independent services. The characteristics of such application perfectly fit the Microservices Architecture pattern.

Microservices are a fast growing trend in cloud-based application development [14, 109]. The traditional monolithic application is divided into small pieces that provide a single service: the full capabilities of the application emerge from the interaction of this small pieces. Microservices are independent from each other and organized around capabilities, e.g. user interface, front-end, etc. Their decoupling allows developers to use the best technology for their implementation according to the task they have to accomplish: the application becomes polyglot, involving different programming languages and technologies.

An application composed of microservices is inherently distributed: it is divided into hundreds of different microservices deployed in a large network infrastructure, that communicate both in a synchronous or an asynchronous way, using REST or a message-based system respectively. Through the use of microservices, the application can scale efficiently, since it is possible to scale only the microservices that are under heavy load, not the entire application.

The microservices architecture embodies the principles of the DevOps
5.3. Technological considerations

movement, promoting the automation of deployment and testing and reducing the burden on management and operations \cite{82}. Several companies in the recent years moved to a microservices architecture: Netflix has been among the first to adopt microservices, followed by SoundCloud and Groupon among the others \cite{6,16,17}.

The use of the Microservices Architecture for the development of a cloud-based application provides several advantages, but at the same time poses new challenges to software developers, both in the creation of a new software and in the migration from a legacy system to a cloud-native architecture \cite{15,18,30}. In particular, we can highlight some issues related to the adoption of the Microservices Architecture pattern:

- **Overhead in operations.** The use of microservices can make difficult the deployment of the application, since it is necessary to deploy a huge number of pieces instead of a single one. The use of containers and tools like Docker Compose can help with this issue \cite{8}. However, the complete management and deployment is still difficult, and operations such as monitoring or logging become harder;

- **Distributed Systems complexity.** The use of microservices imposes to make the application largely distributed in many of nodes. This brings complexity to the system, since requires to address additional problems like the service discovery and load balancing between them. Understanding if the application is running properly is not trivial, and if a problem arises it is difficult to understand its root cause;

- **Polyglot system.** Microservices can ideally be developed with different technologies and languages, since every microservice is independent from the others. This is a plus for the performance of the application, since every service can be built using the technology that is most suited for the task it has to accomplish. However, the integration between microservices and other tools (e.g., monitoring systems, message queues, etc.) can be problematic and lead to chaos in the system.

We think that Self-Adaptation can help developers in the challenges posed by microservices. Operations like deployment and service discovery can be actuated in an automatic way. The use of a Self-Adaptation can greatly reduce the effort in the management of a distributed system like the one imposed by the adoption of microservices: fault-tolerance, service discovery, load balancing, dynamic scaling, resource management can all be handled autonomously using Self-Adaptation. Moreover, the self-adaptive
Chapter 5. Enabling Transparent Adaptation in Complex Systems

capabilities can be introduced externally and integrated in a transparent way into the application, handling the polyglot nature of a microservices application.

These are the motivations that led us to choose cloud-based applications built using the Microservices Architecture pattern as the application domain for our tool.

5.3.3 Adaptation enabler: Docker containers

The adaptation enabler is a virtual container that encapsulates every single service. The development of a specific container for our tool would reduce its applicability on different applications and potentially has a bad impact on its adoption by practitioners. Since in the last years the virtual container technology saw a rapid growth and adoption by the industry, we prefer to adopt the implementation provided by Docker, which is the de facto standard in industry.

Docker containers can run an application as an isolated process on a host machine, including only the application and all its dependencies (the kernel of the Operating System is shared among different containers) and providing to it only the resources it requires \[7, 34, 146\].

Docker containers are different from a fully virtualized system like a Virtual Machine: a Virtual Machine contains a full OS that runs in isolation on physical resources that are virtualized by an hypervisor on the basis of the ones available in the host machine; a Docker container uses the resources available in the host (both a physical or a virtualized one) that are assigned to it by the Docker Engine. The consequence is that Docker containers can share physical resources and are lightweight: it is possible to run multiple containers on the same machine starting them in seconds.

Docker allows developers to implement their application and their services using the technology or language that is most suitable for them. Services deployed in a Docker container can be scaled or replaced just starting or stopping the container running a specific service. Moreover, Docker containers can be deployed in very different settings, from servers in a cloud computing infrastructure to ARM-based IoT devices.

5.4 Self-Adaptation in Microservices Architecture and Docker containers

We identified Docker containers and the Microservices Architecture as central elements in the implementation and application of our tool. Despite
5.4. Self-Adaptation in Microservices Architecture and Docker containers

their recent rise, they have been the subject of some research projects related to Self-Adaptation. In this section we want to provide an overview and state of the art related to the researches involving Self-Adaptation, Microservices Architecture and Docker containers.

5.4.1 Microservices Architecture Self-Adaptation

Microservices present various advantages like encapsulation, maintainability and fine grained management. However, they introduce into the application a bigger complexity due to the fact that the system is composed of many small pieces that should be monitored and managed in order to keep the global application up and running. Even with the use of containers, the management of this kind of large-scale decoupled application is not trivial. This is the reason why Docker containers have been addressed in the research too, proposing solutions for their management and exploiting them for the auto-scaling of cloud applications. Self-Adaptation has been applied to handle this complexity, introducing inside the microservices application running in Docker containers self-adaptive capabilities at various levels.

An architecture for self-managing microservices has been proposed in order to enable the scalable and resilient self-management of microservices in [144]. The authors propose a distributed architecture based on self-managing atomic services and on the election of a cluster leader that takes the decisions and actuates the actions. The management logic is present in each service, so if the leader fails another one can be elected and manage the other nodes. The system is composed of two layers: the local cluster that contains the microservices and one leader, and a composition cluster composed of all the leaders for endpoint discovery across microservices and for leader election. The adaptation logic is internal to each microservice and the management adopts a hybrid approach.

App-Bisect is a tool for the self-healing of cloud applications based on the Microservices Architecture pattern deployed in a production environment [124]. App-Bisect operates like a versioning system for the microservices that are deployed. If there is a loss in performance of the application after an update, App-Bisect is able to revert a specific microservice to a version that ensures the desired performance, keeping the application up and running until a solution for the failure is found. The solution is very interesting but limited to the self-healing of the application, without introducing into the application other self-adaptive capabilities.

Serfnode addresses the problem of service discovery of microservices and it is based on Docker containers and the Serf project [23, 140]. Serfnode
Chapter 5. Enabling Transparent Adaptation in Complex Systems

is an agent that acts as a parent container of the services of the application: each service image has its corresponding Serfnode parent that monitors that service and handles the discovery of the service as well as various types of events. Serfnodes communicate using a gossip protocol, providing a fully decentralized system.

Self-Adaptation has been applied to microservices to address specific issues, like self-healing or service discovery. There is no a complete self-adaptive solution for microservices, which allows to self-manage the application through complex adaptation actions. The architecture proposed in [144] is based on the internal approach, so the adaptation logic should be integrated inside the microservices, preventing it to be applied to applications already developed and deployed. The tool we propose aims to provide advanced self-adaptive capabilities – i.e. auto-scaling, self-optimization, etc. – to a wide range of applications that are not self-adaptive and that are not designed to integrate self-adaptive capabilities.

5.4.2 Docker containers Self-Adaptation

**Autoscaling of web applications** in containers is addressed in [33]. Here, the traditional MAPE loop is enriched with a new planner that consists of a discrete-time feedback controller. The proposed self-adaptive framework is applied to multi-tier cloud-based web applications, managing Virtual Machines and containers to provide a better granularity in the resource management of the application. This let the system enable a coordinated infrastructure and platform adaptation. The self-adaptive framework has been evaluated with 2 different applications deployed in the Amazon Web Services cloud infrastructure, showing the improvement in the usage of resources using containers and in comparison to the autoscaling mechanism provided by Amazon itself. However, the evaluation has been done using a very limited resource pool, i.e., 10 VMs with 1 core in one case and 1 VM with 8 cores in the other one, so it does not represent a realistic setting.

**Smart Brix** is a framework that enables the dynamic evolution of systems composed of containers [133]. Smart Brix is able to ensure that the containers respect the requirements imposed on the system through the use of compensation pipelines composed of self-assembling components that are able to autonomously operate on container. Through the compensation pipelines Smart Brix can validate and verify containers, checking a large amount of heterogeneous containers against a set of evolving requirements that may comprehend vulnerabilities, compliance constraints, and is able to evolve the containers in order to mitigate the issues that emerged from the
5.4. Self-Adaptation in Microservices Architecture and Docker containers

checks.

Docker containers have been exploited to obtain the elastic scaling of the application [80]. Using a multi-objective optimization model to allocate containers on top of Virtual Machines the application can be scaled elastically reducing the consumption of resources. The optimization model works both at the level of Virtual Machines and Docker containers, actuating vertical and horizontal scaling.

DoCloud is an elastic cloud platform that exploits Docker to adapt the web application to the changing workload scaling the Docker containers composing the app [89]. DoCloud integrates a load balancer (HAProxy) and a private Docker registry to store the containers images. The platform uses a hybrid elastic controller that incorporates a proactive and reactive model for scaling the containers. The proactive and reactive models are used for the scale-out, while for the scale-in only the proactive model is used.

Several solutions for the management of a cluster of Docker containers have been proposed in literature [117]. Among the most commonly used tools there are Swarm and Kubernetes [9, 13, 34].

Swarm is the native solution for the clustering of Docker containers [9]. After installing a Swarm agent in every node of the cluster, the user can control them through the manager. The user can start and stop containers letting Swarm decide where to place them according to different strategies (random, bin-packing, spread) that take into account the resources available in the nodes. The user can also specify some affinities between containers, so Swarm can try to place together containers that have an affinity.

Kubernetes is a solution provided by Google for containers orchestration and clustering [13, 34]. Kubernetes handles scheduling of containers on the nodes and manage workloads to ensure that the system meets some user constraints. Containers are grouped into pods and using labels to create logical units for easy management and discovery. Kubernetes offers also a system for failure recovery: using a replication controller a container or machine that fails can be restarted automatically.

These are commercial solutions used in production systems and greatly simplify the management of the application. However, they do not provide autonomic capabilities and the management of the application still depends on user intervention. In general, Docker containers have been mostly exploited to simplify the scaling of the resources. We want to study the role they can have in the application of Self-Adaptation to external applications, exploiting them to enable transparent adaptation in complex systems.
Chapter 5. Enabling Transparent Adaptation in Complex Systems

5.5 Conclusion

The analysis of the design choices at the base of the SELFLET Framework helped in the identification of its advantages – like decentralization, encapsulation of services, emergent behavior, etc. – and its limitations, that we identified as the difficulty in the learning of a new framework by developers, the use of an internal approach for the integration of the adaptation logic and in the management of services inside the containers.

We started from this analysis to design a software tool that presents all the advantages of the SELFLET Framework, but tries to overcome its limitations through the use of an external approach for the integration of the adaptation logic, and its transparent application to the managed application using virtual containers. We proposed the decomposition of a distributed Self-Adaptive System into three distinct components interacting between them: the adaptation manager containing the adaptation logic, the managed element that is the application to manage, and the adaptation enabler that acts as the connection point between the two parts.

The technology to a possible implementation of the tool has been identified in the Microservices Architecture and Docker containers. Microservices Architecture is a pattern for the creation of an application that has all the characteristics needed by our managed element; this makes it a very good application domain for our tool. Docker is the de facto standard in virtual containers and is a perfect implementation for the adaptation enabler.

We then described the relation between Self-Adaptation and both Microservices Architecture and Docker containers, providing an overview and state of the art related to the research efforts in this topic.

In the next chapter we present GRU, an actual implementation of the proposed tool that can bring Self-Adaptation to applications built using the Microservices Architecture pattern and deployed in Docker containers. We evaluate GRU in Chapter 7, validating our ideas using a realistic case study.
CHAPTER 6

GRU: Bringing Self-Adaptation to Microservices Architecture

We presented the design of a tool that can be applied in a transparent way to a distributed system to make it self-adaptive, describing its components – the adaptation manager, the managed element and the adaptation enabler – and providing also an overview of the technologies that may be used for its implementation. However, we need to evaluate the proposed tool with a concrete implementation to validate it.

In this chapter we present GRU, a tool capable of bringing self-adaptation to cloud-based applications developed according to the Microservices Architecture pattern and deployed inside Docker containers. GRU is a possible implementation of the tool described in the previous chapter. The adaptation manager is implemented using GRU-Agents, autonomous agents deployed in every node of the distributed system that implement the MAPE-K loop. GRU-Agents interact to share information and manage the microservices application that represent our managed element. Docker containers act as the adaptation enabler, being the connection point between the GRU-Agents and the microservices running inside the containers: in this way the GRU-Agents are totally decoupled from the microservices application, and
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

can be applied potentially to any distributed system deployed inside Docker containers.

Section 6.1 provides an overview of a system that uses GRU as the adaptation manager of the application. The Architecture of GRU is then described in Section 6.2 where we present all the components of a GRU-based system and the way the components interact between them. Section 6.3 is dedicated to a deep focus on the GRU-Agent, providing details regarding the internal components of the agent and the part that are configurable by the user. We briefly describe the technical implementation of GRU in Section 6.4, while in Section 6.5 we discuss its limitations. Section 6.6 concludes the chapter.

6.1 Overview of the GRU system

GRU is a tool able to bring self-adaptive capabilities to applications developed as composition of microservices running in Docker containers. Since GRU is designed to be applicable to virtually every application developed as microservices running in Docker containers, it uses an external approach in order to make the application self-adaptive. This means that the self-adaptive capabilities are not part of the application itself, but the adaptation manager is independent from the application and interact with it through an interface. This makes the application easier to design and develop, since the developers do not have to deal with the self-adaptive part. GRU is composed of independent agents, called GRU-Agents, deployed on each host running the application: GRU-Agents interact with the Docker Daemon making the application self-adaptive. Once the application is developed and deployed in a cluster of nodes, the user just have to configure the GRU-Agents, provide a description of the microservices (in the form of configurations files we call µService-Descriptors, described in Section 6.2), and let them manage the application. Figure 6.1 depicts an overview of a GRU system, where an application composed of three microservices running in Docker containers is deployed in a cluster of two nodes, where GRU-Agents are running.

GRU can ensure that the application is always up and running and that meets some quality constraints, such as the Maximum Response Time of the microservices (i.e. the maximum time interval between the arrival of a microservice request and a its response). GRU is open source and available on Github.

Differently from a centralized approach, where there is a centralized manager that operates the decision making process, or a hierarchical ap-

https://github.com/elleFlorio/GRU
6.1. Overview of the GRU system

Figure 6.1: Overview of the GRU system, with two nodes.

proach, where the system is layered and sets of peers are managed by a leader that takes all the decisions, GRU is based on a fully decentralized feedback loop. GRU-Agents implement the feedback loop internally and can decide about the action to actuate in an independent way. The decision is made according to two kind of information: (i) the internal state of the GRU-Agent, i.e, the data gathered from the Docker containers running the microservices of the application in the same host of the GRU-Agent; (ii) the data coming from a subset of the total number of peers. GRU-Agents work with partial information: they do not have a knowledge about the state of
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

the whole system, but they can have information about a part of it. During every loop iteration each GRU-Agent chooses a predefined number of peers, and ask for the data of that peers. The chosen peers are selected randomly at each iteration among the ones in the cluster: in this way each GRU-Agent has a different point of view of the system at every iteration.

Features

In this section we describe the main features of GRU, in order to highlight what makes GRU different from other self-adaptive systems related to microservices applications that have been presented in literature (see Chapter 5).

Decentralized

The Decision on the actions to actuate on the application is taken by each GRU-Agent in an independent way, and the self-adaptive capabilities of the application emerge from the local decisions and actions of each GRU-Agent. There is not a central management layer that collects all the statistics of the system, and the autonomic loop itself is decentralized. Using this approach GRU presents no single point of failure, increasing the fault tolerance and availability of the system.

Work with partial information

GRU-Agents decide the best action to actuate according to their internal state and the information coming from a subset of the total number of peers that changes every time. During each self-adaptive loop iteration each agent has a partial view of the system and actuate the appropriate action on the application using this partial information and a probabilistic approach: in this way there is no information exchange overhead.

Highly customizable

GRU is designed to let the user define the metrics and the analytics he wants to compute about the state of the system. In this way GRU can be easily adapted to every application and to the user needs.

Plug & play

GRU is designed to be easily “pluggable” to every application designed as a composition of microservices running in Docker containers. The application needs only to send GRU statistics that can be used to understand
the best action to actuate in order to keep the application up and running. Developers do not need to implement a complex logic to make the application self-adaptive, they just need to provide to GRU the data it requires and focus on the functional part of the application. This can help to make Self-Adaptive Systems more appealing to developers.

6.2 GRU Architecture

GRU is composed of a set of GRU-Agents deployed in nodes. Each node belongs to a cluster. The information about clusters, nodes, and the configuration of GRU-Agents is stored in an external repository.

Nodes represent the hosts where the GRU-Agents and the microservices are running. Each node has a unique name and ID and it is registered to one (and only one) cluster. The user can set a property on each node called base-services property: it represents the set of the microservices that should be running in a specific node, and GRU ensures that at least one instance of these microservices will be running on the node.

Clusters are set of nodes where GRU-Agents are running. Each cluster presents a unique name and ID. Each GRU-Agent should register the node where it is running to a cluster in order to be visible to others and to exchange information. Nodes can belong to one cluster only.

The external repository has a dual role: (i) shared configuration storage (ii) agent discovery system. The shared configuration of the cluster is stored in the repository. The external repository acts also as a discovery system for the GRU-Agents: when an agent starts, it registers itself to the external repository with its ID and address. The record of each agent has a predefined time to live (TTL), so if an agent fails and cannot confirm that is active it is removed from the list. GRU-Agents can query the external repository to get the list of active peers.

6.2.1 Cluster configuration

The configuration of the cluster can adapt GRU to the application to manage according to the user needs. It is composed of the agent configuration, the µService-Descriptors of each service, the policy configuration and the analytics defined by the user.

Agent configuration

The agent configuration is shared among all the GRU-Agents and contains all the information needed by the agent to operate and to communicate
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

with external services. It contains the parameters needed to connect to the Docker daemon, the parameters related to the feedback loop such as the loop time interval and the strategy to actuate, the time interval for the communication and the number of peers to reach, and the configuration of the discovery service of the application and an optional monitoring system that can be used to monitor the state of GRU.

μService-Descriptors

μService-Descriptors are abstractions that model the microservices of the application, and they are used by GRU-Agents to understand how to properly manage each microservice. μService-Descriptors present three kind of information: the service information, the user-defined information and the docker configuration.

The service information includes all the general information about the service, such as its name, its type (e.g., webservice, database, etc.) and the Docker Image it is associated with (the docker image is like a blueprint of a docker container used to create a new one). Despite the information about the type of the microservice is not used at this stage of development, it can be exploited to reason about the composition of the application and to apply specific actions to specific types of microservices.

The user-defined information comprehend some constraints that the user wants to impose on the microservice (e.g., the Maximum Response Time of the service) and the analytics that should be computed (a more detailed description of the analytics will follow).

The Docker configuration contains all the parameters needed to properly create a Docker container running the microservice, such as the resources needed (number of CPUs, amount of memory), environmental variables and the parameters that should be passed to the microservice when it is started.

Analytics

Analytics are values between 0 and 1 that can be computed according to an equation provided by the user. Every analytics has a name, an expression representing the computation to be done and the metrics and constraints that are present in the expression as variables. Every microservice in its descriptor present the analytics that should be computed for that specific microservice, as well as its constraints.
6.2. Gru Architecture

Policy Configuration

The policy configuration contains the parameters needed by the implemented policies, such as their threshold and the metrics and values to be taken into consideration in the computation of the weight of the policy. In the policy configuration the user can also enable or disable every single policy through a flag. Policies are described in details in Section 6.3.

6.2.2 Automatic resource management

Gru is able to manage the resources of the cluster in an automatic way. Currently it is able to assign to microservices the available CPUs according to the number set in the µService-Descriptor of the microservice: using Docker the user must set the index of the CPUs assigned to a container, while Gru allows the user to simply set the number of CPUs the container requires and Gru takes care of allocating to the container the available ones. However, if the user wants to run a container on specific CPUs, Gru will use that CPUs if they are available. This is just a basic feature, and we are going to study and implement more sophisticated policies for resource management and for resource optimization inside the cluster.

6.2.3 Microservices Registration

Microservices don’t have to register themselves to an external discovery service, but Gru-Agents will handle the process on their behalf. Gru can register microservices to a discovery service keeping them alive with a TTL set by the user in the agent configuration.

6.2.4 Gru-Agents Interactions

Gru-Agents interact between them and with the Docker containers running the microservices of the application (Figure 6.2). The communication between other peers is synchronous and happens using REST. The data exchanged by Gru-Agents are the local data computed by the Analyzer component (more on this in the following section).

Gru-Agents interact with the Docker containers in its node through the Docker Daemon running in background. Gru-Agents exploit the API provided by Docker to query the Docker Daemon about the state of the containers: it is possible to retrieve low-level information about the consumption of resources (CPU and memory usage, storage, I/O operations, etc.), as well as the specific properties of each container, such as the number of CPU assigned, the memory limit, the network interface, etc. Gru-Agents can
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

Figure 6.2: Interactions of the GRU-Agent.

also read the logs of the microservices that are exposed by the container, accessing higher-level information. This information is used to understand the state of a container and the total consumption of resources of the node, as well as the state of the microservices inside the containers.

6.3 GRU-Agents

GRU-Agents are deployed in each node of the cluster and interact with the Docker Daemon to manage the containers running the microservices of the application.

When GRU-Agents join a cluster, they download from the external repository the µService-Descriptors, the policy configuration, the analytics and their configuration. GRU-Agents present different components that interact between them and that are depicted in Figure 6.3. The REST APIs allow the agent to communicate both with the external environment and other GRU-Agents.

The Internal Storage is used to store the cluster data retrieved from other GRU-Agents through the REST APIs as well as the local data computed by the agent itself during the feedback loop.

The Communication Manager exploits the REST APIs to connect to other agents and get their data. The communication happens with a time interval that is defined in the agent configuration. First, the external repository is contacted to get a subset of the running peers that is chosen randomly. The random selection ensures to have a broad view of the state of the system as the number of iterations increases. Once the subset of peers is chosen, the Communication Manager gets the data of the peers through the REST APIs and stores them in the Internal Storage. The data shared
6.3. Gru-Agents

Figure 6.3: Internal components of a Gru-Agent.

between agents are called Shared Data and comprehend the information on the CPU and memory usage, the analytics defined by the user as well as the last chosen policy computed locally. These data are merged computing the mean of each value and stored in the Internal Storage as Cluster Data; this represents the partial view of the system, and the decision about the action to actuate is based on this information. The use of the partial view of the system lets each Gru-Agent make decisions on a more solid base than the local information only.

Gru-Agents exploit the MAPE-K feedback loop to manage the application. The loop is composed by four elements: Monitor, Analyzer, Planner and Executor, while the Knowledge Base is represented by the Internal Storage.

Monitor

The Monitor component interacts with the local Docker Daemon to get statistical information about the resource usage of each Docker container (e.g., CPU usage, memory usage), as well as every container-related events (e.g., start/stop of a container). The Monitor can parse the logs of the con-
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

tainer to get information about the internal state of the microservice: if the microservice logs its response time or the number of requests it is receiving, the monitor is able to retrieve this information. The Monitor can also receive data by the microservices through the REST API that exposes an endpoint for this purpose. The Monitor is composed of two parts: (i) a “live” part that is constantly running and monitoring the containers and (ii) a part that is activated at every iteration of the feedback loop and prepares the data for the Analyzer component.

Analyzer

The Analyzer component receives the data coming from the Monitor and elaborates them to obtain the analytics values (i.e., values between 0 and 1 that are then used by the Planner component for the decision making process). The analytics values comprehend the CPU and memory usage, as well as the user-defined analytics (see Section 6.2.1). The Analyzer merges the computed local data to the Cluster Data gathered by the Communication Manager and stores the results of its computation in the Internal Storage. Once the data are computed they are sent to the Planner component.

Planner

The Planner component decides what policy should be actuated as the result of the feedback loop iteration. The decision is based on the Cluster Data computed by the Analyzer, i.e. the partial view of the whole system. The basic elements of the decision making process are the policies and the strategies. Policies are rules that triggers some actions that are actuated on the containers, while strategies are algorithms that choose a policy among a set of weighted ones.

Policies have a name that identifies the policy, a list of actions that needs to be actuated to satisfy the policy, a target service for the actions and a weight. The weight is a value between 0 and 1 that is computed according to an equation that is different for each policy. Policies can be enabled or disabled in the policy configuration. The Planner creates a weighted policy for every microservice to manage, resulting in a list of $P \times M$ weighted policies, where $P$ is the number of enabled policies and $M$ is the number of microservices to manage. Currently there are four policies implemented: the scale-in policy, the scale-out policy, the switch policy and the no-action policy.

The scale-in policy triggers the Stop and Remove actions, which stop a container running the target microservice and remove it from the node
6.3. **Gru-Agents**

freeing the resources, respectively. The weight $w_{\text{policy,ms}}$ of the policy for each microservice $ms$ is computed according to Equation \[6.1\]:

$$w_{\text{policy,ms}} = \frac{\sum_{i=0}^{n_{\text{analytic}}} w_{\text{analytic},i}}{n_{\text{analytic}}} \tag{6.1}$$

where $n_{\text{analytic}}$ is the number of the analytics the policy should consider in the computation of its weight, and $w_{\text{analytic},i}$ is the weight of every analytic that is computed according to Equation \[6.2\]:

$$w_{\text{analytic}} = 1 - \frac{\min(v_{\text{analytic}}, \text{thr}_{\text{scale-in}})}{\text{thr}_{\text{scale-in}}} \tag{6.2}$$

where $v_{\text{analytic}}$ is the value of the analytic, and $\text{thr}_{\text{scale-in}}$ is the threshold for scale-in defined in the policy configuration. In the case that a microservice has only one running instance and is in the base-services set of the node, the scale-in policy is not evaluated and its weight is set to 0.

Using Equation \[6.1\] and Equation \[6.2\] the weight of the policy is proportional to how much the load of the microservice is below a threshold defined by the user: the analytics used to compute the weight of this policy should be related to the load that the microservice is facing (e.g., the response time of the service, the resource consumption, etc.).

The **scale-out** policy triggers the \textit{Start} action, which starts a new instance of a container running the target microservice. The \textit{Start} action starts a container if in a stop state, or creates and starts a new one otherwise. The weight of the scale-out policy is computed according to the same equation of the scale-in one (see Equation \[6.1\]). However, the $w_{\text{analytic}}$ value is computed differently and according to Equation \[6.3\]:

$$w_{\text{analytic}} = 1 - \frac{\max(v_{\text{analytic}}, \text{thr}_{\text{scale-out}}) - \text{thr}_{\text{analytic}}}{1 - \text{thr}_{\text{scale-out}}} \tag{6.3}$$

where $\text{thr}_{\text{scale-out}}$ is the threshold for scale-out defined in the policy configuration. In case there are not enough resources to start a new instance of a microservice, the scale-out policy is not evaluated and its weight is set to 0.

This policy is the dual of the scale-in one, so Equation \[6.1\] and Equation \[6.3\] have been chosen to compute a weight that is proportional to how much a service is overloaded, taking as a reference a threshold defined by the user. For this reason, the analytics involved in the computation of the weight of this policy should be related to the load that the microservice is facing, as in the scale-in policy.
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

The **switch** policy allows to switch a running microservice with another one that is not running in a single iteration. This policy triggers first the Stop and Remove actions on a running container of a microservice, then triggers the Start action on the container of a different microservice. This policy is actuated only if the node has not the resources needed to start a new microservice, but needs to stop another one in order to obtain such resources. The switch policy is computed on pairs of microservices, in order to understand if one service should be replaced by another. First, microservices are divided into running ones and inactive ones, then a switch policy is created for each pair running-inactive assigning to it a weight. The weight of this policy is computed according to the following equation:

\[
\begin{align*}
    w_{\text{policy,pair}} &= \text{Max} \left( 0, \sum_{i=0}^{n_{\text{analytic}}} w_{\text{analytic,i}} n_{\text{analytic}} \right) \quad (6.4)
\end{align*}
\]

where \( w_{\text{analytic,i}} \) is computed according to Equation 6.5

\[
\begin{align*}
    w_{\text{analytic}} &= \text{Min} \left( 1, \frac{\text{dist}_{\text{analytic}}}{\text{dist}_{\text{max}}} \right) \quad (6.5)
\end{align*}
\]

The value \( \text{dist}_{\text{max}} \) is the maximum distance that can occur between the value of two analytics and it is defined in the policy configuration. The Equation 6.6 is used to computed the distance between two metrics, i.e. \( \text{dist}_{\text{analytic}} \):

\[
\begin{align*}
    \text{dist}_{\text{analytic}} &= v_{\text{analytic,inactive}} - v_{\text{analytic,running}} \quad (6.6)
\end{align*}
\]

where \( v_{\text{analytic,inactive}} \) and \( v_{\text{analytic,running}} \) are the value of the same analytic for the inactive and the running microservices respectively. The computation of the switch policy is evaluated only between microservices that share the same analytics.

The equations used to compute the weight of the switch policy have been studied to express the difference on the load that two microservices are facing: the weight of this policy computed between two microservices is proportional to a maximum distance that the user imposes between the load of the two microservices.

The **no-action** policy simply does not trigger any action. It is weighted according to Equation 6.7

\[
\begin{align*}
    w_{\text{noaction}} &= 1 - \max(\text{policiesWeights}) \quad (6.7)
\end{align*}
\]

The value \( \text{policiesWeights} \) is the set of weights of all the other computed policies, so the weight \( w_{\text{noaction}} \) is computed as the difference between one and the maximum value computed for the other policies.
6.3. Gru-Agents

Algorithm 7 Dummy Strategy Algorithm

1: \textbf{maxWeight} \leftarrow 0
2: \textbf{for } p \in \textbf{policies} \textbf{do}
3: \textbf{if } p\text{.weight} > \textbf{maxWeight} \textbf{then}
4: \quad \textbf{chosenPolicy} \leftarrow p
5: \quad \textbf{maxWeight} \leftarrow p\text{.weight}
6: \textbf{end if}
7: \textbf{end for}
8: \textbf{return chosenPolicy}

The no-action policy should be actuated as an alternative to other policies when they are not required. Using Equation 6.7 we can assign to this policy a weight that depends on the ones computed for the other policies, and that expresses that the system does not require any adaptation action.

Once a weight is assigned to each policy for each service, policies are analyzed according to a specific strategy. Strategies are algorithms used to choose the right policy to actuate among the list of the available ones taking into account their weight. The system can have several strategies implemented, but only one active in the same cluster. This means that there cannot be two agents with a different active strategy at the same time in the same cluster. The current release of GRU implement two strategies: the dummy-strategy and the probabilistic-strategy.

The dummy-strategy simply chooses the policy with the highest weight and uses Algorithm 7 for the selection of the policy to actuate.

However, locally choose the policy with the highest weight could not be the best strategy, leading to local optima that can produce undesired behavior of the entire system. Using this strategy, all the GRU-Agent would share the same behavior, potentially over-reacting to a problem and making the system unstable (e.g., actuating too many scale-out policies during a spike in the workload).

The probabilistic-strategy is based on a probabilistic computation to choose the policy to actuate. Using a probabilistic approach, the Planner can avoid local optima in the selection of the policy to actuate. Algorithm 8 is used for the selection of the policy.

The strategy acquires as input an array of weighted policies policies and shuffles it. It computes the totalWeight as the sum of the weights of all the policies and uses this in the next steps to normalize all policy weights. Moreover, it chooses randomly a threshold. It then checks for each policy in policies if its normalized weight is greater than the threshold. If this is the case, it then selects that specific policy for execution. Otherwise it looks
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

Algorithm 8 Probabilistic Strategy Algorithm

1: policies ← Shuffle(policies)
2: totalWeight ← \( \sum_{p \in \text{policies}} p.w\text{eight} \)
3: threshold ← rand\((0, 1)\)
4: delta ← 1
5: index ← 0
6: for \( p \in \text{policies} \) do
7: if \( \frac{p.w\text{eight}}{\text{totalWeight}} > \text{threshold} \) then
8: return \( p \)
9: else
10: if \( (\text{threshold} - \frac{p.w\text{eight}}{\text{totalWeight}}) < \text{delta} \) then
11: delta ← threshold - \( \frac{p.w\text{eight}}{\text{totalWeight}} \)
12: index ← index \( p \)
13: end if
14: end if
15: end for
16: return \( \text{policies}[\text{index}] \)

for the remaining policies in the array. In the search, to address the case in which none of the policy normalized weights passes the threshold, it keeps track of the difference between such weights and the threshold, storing the index of the policy that is closest to the threshold in \( \text{index} \). Thus, it can select, in the end, the policy with weight closest to the threshold. This algorithm is a slight variation of the one adopted by the SELFLETS that is described in Chapter 3 and is based on the same probabilistic approach that uses the principle of noise \([77, 110]\). This approach has been used in literature in various works, and represents a good strategy to avoid local optima during the adaptation phase \([43, 58, 110, 147]\).

The policies and the strategies described here are currently in a limited number and hard-coded inside the system. This is a limitation of our prototype, as discussed in Section 6.5. We plan to expand the available policies and strategies allowing the user to define its own equations and algorithms using specific configuration files that will be read by GRU-Agents.

Once the policy has been chosen, the Planner component creates a plan that contains the policy to execute and the target service. The plan is then sent to the Executor component.

Executor

The Executor component actuates the actions of the policy of the chosen plan on the target microservice. Before executing an action, the Executor chooses the resources to allocate, if needed, and creates the configuration
6.4 System implementation

GRU is implemented using the Go programming language and works in a Linux environment. GRU has few dependencies in order to be deployed: a working instance of an etcd server and an influxDB database. Etcd is a distributed key-value store and it is used by GRU as the external repository and for GRU-Agents discovery. InfluxDB is a storage system for time-series data and it is used by GRU as the external monitoring system where the metrics collected by the GRU-Agents are sent. InfluxDB is compatible with Grafana, a data visualization tool that allows to monitor GRU in real time. GRU is implemented without using any specific cloud vendor API; this means that GRU is cloud-independent and can run in the cloud infrastructure of every vendor and even in a multi-cloud environment. GRU agents communicate through a REST interface between them and with the environment.

GRU implements three commands that can be executed: create, join, and manage. The Create command is used to create a cluster and the related folder tree in the etcd server. The user must specify the name of the cluster. The Join command allows GRU-Agents to join a specific cluster. The user must specify the cluster to join and the etcd server address. The User can assign a name to the node, otherwise GRU assigns a random generated name to it (the name is generated coupling a random adjective from a list to a random name from another list). During the join operation the GRU-Agent connects to the etcd server to register itself to the list of the nodes available in the cluster, reads the configurations of GRU-Agents, node and µService-Descriptors, and initializes all the components of the system (e.g., storage system, connection to the Docker Daemon, etc.). Once the configuration is finished the GRU-Agent starts the REST-server waiting for commands. The Manage command starts a command-line client that allows the interaction

---

[https://github.com/coreos/etcd](https://github.com/coreos/etcd)
[https://github.com/influxdb/influxdb](https://github.com/influxdb/influxdb)
[https://github.com/grafana/grafana](https://github.com/grafana/grafana)
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

with GRU-Agents. The user can list clusters, the nodes of a cluster and check the µService-Descriptors and the configuration of the GRU-Agents. Through the command-line client the user can start microservices in a specific node and set properties on the nodes such as the base-services. Once everything is set, the user can start the GRU-Agents either in specific nodes or in all the nodes.

Example deployment

We want to provide an idea of the steps required to deploy and run GRU. Let’s suppose the user wants to deploy GRU in a set of machines in a cloud infrastructure. We assume that the user has already developed an application as a set of microservices running in Docker containers, and she has already prepared the µService-Descriptors according to the microservices she wants GRU to manage. The first step is to set up a machine with an instance of an etcd server and an instance of influxDB. We assume that the address of etcd server is exported in some environment variable in every machine, so it is not necessary to specify it in every command.

The user can execute the command gru create my-cluster to create a cluster called my-cluster. After the creation of the cluster, the user must upload the configuration of the GRU-Agents and the µService-Descriptors into the etcd server, as well as the policy configuration and the user-defined analytics. To handle this operation in an easy way we are working on a configuration manager that can run as a web application. Once the configuration has been uploaded in the etcd server, the machines can join the cluster as node using the command gru join my-cluster. This operation can be carried out in an automatic way creating a machine image that executes the join command at the boot of the machine itself.

The user can execute the gru manage command in a machine of his choice (even his laptop), check the list of nodes in the cluster (> list nodes), set up some properties like the base-services of a node (> set node node-A base-services serv1 serv2), start a microservice (> start service serv1 node-A) and then start the GRU-Agents (> start agent all). GRU is running and the user can check the status of the system connecting an instance of Grafana to influxDB and preparing a dashboard with the data she wants to monitor.
6.5. Limitations

GRU presents some limitations that we will address during its future development.

Automatic resource management provides a higher level of automatism compared to the Docker one. However, as said in Section 6.2, it is still in a prototype stage and can be improved with the study of algorithms for the optimization of resources usage and allocation.

Currently GRU is able to actuate only three self-adaptive policies: scale-in and scale-out of microservices and swap of microservices. This enables the auto-scaling of the application according to its usage and constraints. However, we would like to add more self-adaptive policies to the list of the available ones.

We are now working on the implementation of two more policies to change the implementation of the microservices according to the load they are facing – i.e., the same Change Implementation policy implemented in the SELFLET Framework (see Chapter 3) – that we could not integrate in this version of the tool due to time constraints.

Other policies may include the migration of microservices from one host to another, the “warm” upgrade of microservices to a new release, and the choice of different versions of the same microservice according to certain criteria. All these policies can be actuated on any GRU-agnostic microservice-based system by exploiting the features offered by the Docker containers.

GRU-aware policies, such as the possibility to modify the internal behavior of an application, would require GRU to send commands to the microservices of the application in the form of a message. This would imply the definition of proper guidelines for application developers to enable the creation of an endpoint to receive commands from the GRU-Agents, making GRU able to actuate self-adaptive actions that changes the functional part of the application. Such extension would allow GRU to properly handle also the case of scaling of stateful microservices. Despite microservices in Docker containers are stateless by design, there can be situations where a microservice needs to be stateful and save its state in some storage in order to operate. By developing the communication mechanism between GRU-Agents and microservices, we could delegate to these last ones the responsibility to externalize and maintain the consistency of their state when needed for scaling or any other adaptation purpose.

In this way the “plug-and-play” nature of GRU would not be affected, and the user would be able to choose between two levels of self-adaptation,
Chapter 6. GRU: Bringing Self-Adaptation to Microservices Architecture

i.e., the agnostic one, based exclusively on the exploitation of Docker features, and the GRU-aware one. We are currently studying this scenario for a future implementation in our tool.

6.6 Conclusion

This chapter has been dedicated to the presentation of the GRU tool. GRU is a prototype implementation of the software tool defined in Chapter 5. We applied GRU to the application domain of microservices applications running in Docker containers. We presented the main features of our tool and all the components that form its architecture. GRU is a MAS-based system, so we went deep into the details related to the GRU-Agents and their internal structure.

We highlighted also the limitations of our prototype that could be the focus of future work. We still need to evaluate GRU in order to prove not only the effectiveness of the tool itself, but also the validity of the design we proposed. This evaluation is the topic of the next chapter.
CHAPTER 7

Evaluation of the GRU prototype

This chapter presents the evaluation of GRU and the discussion on the results obtained. In the previous chapter we described all the characteristics of GRU and its architecture. We applied GRU to the same Video-Provisioner application used for the evaluation of the SELFLET Framework, and which has been implemented as a microservices application running in Docker containers. The use of the same application in the tests allows to make a general comparison between the performance of the two systems, that we will present during our discussion on the results.

The objectives of our evaluation are the following:

- **OBJ1**: show that the system can limit the violations of the maximum response time imposed as a constraint on the microservices application;
- **OBJ2**: show that the system self-adapts according to the workload minimizing the waste of resources;
- **OBJ3**: Validate the analytics designed for the tests;
- **OBJ4**: Compare the results obtained by GRU and the SELFLET Framework in the test with the bimodal workload.

95
Chapter 7. Evaluation of the GRU prototype

GRU has been evaluated with two tests, designed to verify the capability of GRU to handle sudden changes in the workload, and its capability to handle a realistic workload. The second workload is exactly the same one used in the evaluation of the SELFLET Framework: in this way we can directly compare the performance of the two systems in handling a realistic traffic coming to the application. We discuss the obtained results to understand if GRU can effectively introduce self-adaptive capabilities in a transparent way inside a microservices application. We also point out the issues of the current implementation and possible improvements that can be the subject of further studies.

This chapter is organized as follows: Section 7.1 describes the Video-Provisioner case study that has been implemented as a microservices application. We then present the configuration of the system – i.e., the hardware setup, parameters of the system, analytics used by GRU, etc. – used during the tests in Section 7.2. The results of the evaluation are presented in Section 7.3 while Section 7.4 is dedicated to the discussion about the results and the comparison to the SELFLET Framework. Section 7.5 concludes the chapter.

7.1 Case study

The case study we adopted for the evaluation of GRU is the same one used for the evaluation of the SELFLET Framework described in Chapter 4. The difference is that the Video-Provisioner application is now implemented using the Microservices Architecture pattern. The application is then composed of eleven microservices, each of them executing a specific task.

The execution flow of the application is depicted in Figure 7.1. In order to simulate the possible choices of the user, each microservice has a specific probability (P) to send a message to another one, represented as the value next to each arrow.

The application starts with the videoprovisioner microservice, that receives the requests and manages the session for the user. The application first checks if the video is available locally, otherwise downloads it (checkiflocallyavailable, downloadvideo). Once the video is available, the application can operate modifications on it according to the user’s choices (from scalevideo to bitdepthreduction). The last step are the operations on subtitles (from checkifsubtitlesavailable to applysubtitle), that are executed if requested by the user. Once the video is ready, the user is notified and can download it from the application.
7.1. Case study

CASE STUDY

Each microservice is implemented using the Go programming language and deployed in a Docker container. The execution of a request is simulated keeping busy the CPU for an amount of time (job-time) that is computed according to an exponential distribution with \( \lambda \) representing the expected demand \( (D) \) of the microservice (see Table 7.1). The value of the \( \lambda \) has been chosen taking into account the type of microservice to simulate. The only exception is the scalevideo microservice, that has a demand derived from statistical data about the video scaling process obtained in a previous research [119]. The requests are processed in series to simplify the application and to have a better control on its execution.

Each microservice presents a destination (the next microservice in the execution flow) and a probability to send a message to that destination. In this way we can simulate the possible choices made by the user about the operations to actuate on the requested video. In case more than one instance of the destination is available, the microservice balance the load among the available instances using a round-robin algorithm.

Figure 7.1: Execution flow of Video-Provisioner application.
Chapter 7. Evaluation of the GRU prototype

Table 7.1: Microservices parameters. \( D \) is the demand of a microservice, while \( P \) is the probability to be executed.

<table>
<thead>
<tr>
<th>Service Name</th>
<th>( D )</th>
<th>( P )</th>
<th>MRT</th>
<th>MRPM</th>
</tr>
</thead>
<tbody>
<tr>
<td>applySubtitles</td>
<td>600</td>
<td>0.2</td>
<td>1200</td>
<td>110</td>
</tr>
<tr>
<td>bitDepthReduction</td>
<td>1300</td>
<td>1.0</td>
<td>3900</td>
<td>50</td>
</tr>
<tr>
<td>checkIfAvailable</td>
<td>1200</td>
<td>1.0</td>
<td>2400</td>
<td>55</td>
</tr>
<tr>
<td>checkIfSubtitlesAvailable</td>
<td>700</td>
<td>0.2</td>
<td>2100</td>
<td>95</td>
</tr>
<tr>
<td>checkIfSubtitlesExists</td>
<td>1500</td>
<td>0.95</td>
<td>3000</td>
<td>45</td>
</tr>
<tr>
<td>downloadVideo</td>
<td>1500</td>
<td>0.95</td>
<td>4500</td>
<td>45</td>
</tr>
<tr>
<td>getDefautSubtitle</td>
<td>1200</td>
<td>0.9</td>
<td>2400</td>
<td>55</td>
</tr>
<tr>
<td>scaleVideo</td>
<td>28500</td>
<td>1.0</td>
<td>85500</td>
<td>3</td>
</tr>
<tr>
<td>transcodeVideo</td>
<td>900</td>
<td>1.0</td>
<td>1800</td>
<td>75</td>
</tr>
<tr>
<td>translateSubtitleToLanguage</td>
<td>500</td>
<td>0.3</td>
<td>1500</td>
<td>135</td>
</tr>
<tr>
<td>videoProvisioner</td>
<td>500</td>
<td>1.0</td>
<td>1000</td>
<td>135</td>
</tr>
<tr>
<td>Application</td>
<td>34100</td>
<td>-</td>
<td>102302</td>
<td>-</td>
</tr>
</tbody>
</table>

The requests coming to the system are registered with a unique ID in an external key-value store. This enable a microservice \( B \) to respond to any of the active instances of the microservice \( A \) that sent a request to \( B \), bringing more flexibility to the application where instances are turned on and off dynamically.

The microservices of the application logs the job-time of every request and the number of requests that they received every minute (RPM). This information is exploited by GRU to manage the application. The microservices communicate with an external monitoring service sending statistical data about their job-time and the RPM. The microservice videoprovisioner sends to the monitoring service also the response time of every request, that represent the response time of the application. The monitoring system is used only for debugging purposes and to check the status of the system during the tests.

7.2 System Configuration

The cluster for the tests has been created on PoliCloud\(^1\) the private cloud infrastructure of Politecnico di Milano. The cluster consists of 28 nodes (gru-node) running the GRU-Agents and the Microservices plus one server used for the deployment of the external repository and the experimental

\(^1\)http://policloud.polimi.it
7.2. System Configuration

infrastructure (*main-node*). The total number of nodes and their com-
putational capability is limited by the available resources provided by the cloud
infrastructure.

Every gru-node offers 2 CPUs and 1GB of memory, while the main-
node is powered by 4 CPUs and 8GB of memory. Despite GRU is poten-
tially able to handle the dynamic creation of nodes that may join or leave
the cluster, for simplicity all the nodes are preallocated. We deployed one
active instance of every microservice belonging to the application in one
different server, except for the *scalevideo* microservice that have five
active instances as default. This represent a reasonable bootstrap for the
system, since the more demanding microservice is the *scalevideo* one.
GRU-Agents run inside a Docker container with limited resource access –
i.e., cpu-shares set to 256 and maximum memory set to 512Mb – to reduce
the impact of the agent on the available resources for the microservices.

In the main-node has been deployed an instance of the etcd\(^2\) server as
the external repository used by GRU, as well Apache Jmeter\(^3\) for the gen-
eration of the traffic, and InfluxDB\(^4\) (a time-series data storage) to store the
statistical data about the status of the system. Grafana web service has been
used for the real-time visualization of the data stored inside InfluxDB.

**GRU configuration**

GRU has been configured using the same parameters for both the test. The
configuration of GRU comprehends the agent configuration, the \(\mu\)Service-
Descriptors of the microservices, the analytics and the policies configura-
tion.

**Agent configuration**

The time interval for the feedback loop of every agent has been set to 120
seconds. The value has been chosen taking into account the job-time and
the constraints about the Maximum Response Time of the microservices
composing the application.

The maximum number of peers to communicate with has been set to
5. This value has been chosen according to the number of nodes in the
cluster to avoid the communication with all the peers and to create a useful
partial view of the system.

The strategy used in the cluster is the probabilistic one described in
Section 6.3

\(^2\)https://github.com/coreos/etcd
\(^3\)http://jmeter.apache.org
\(^4\)https://influxdata.com/time-series-platform/influxdb/
Chapter 7. Evaluation of the GRU prototype

µService-Descriptors

µService-Descriptors are created for every microservice composing the application. Every µService-Descriptor has the information about the microservice and the parameters needed to create a new instance of the specific microservice.

The analytics to compute for every service are the response-time-ratio and the utilization (described in the following paragraph).

The constraints imposed on every service are the Maximum Response Time (MRT) and the Maximum number of Requests Per Minute (MRPM) the microservice can handle. The MRT has been defined as two to three time the demand of the microservice, while the MRPM value has been chosen according to the number of requests the service can satisfy in a minute considering the demand plus the 10% of that value. The MRT and MRPM values for every microservice can be seen in Table 7.1. The demand of the entire application is obtained as the sum of the demand of each microservice multiplied by its probability of execution we defined in the previous section (Equation 7.1), while the MRT of the application is computed as three times its demand.

\[
D_{app} = \sum D_{ms} \times P
\]  

(7.1)

Analytics

The analytics defined are the response-time-ratio and the utilization. The response-time-ratio is defined as the ratio between the average job-time of a microservice and its MRT defined in the µService-Descriptor, so it is computed with Equation 7.2, where \(\text{jobtime}_{avg}\) is the average job-time monitored for all the known instances of the microservice.

\[
\text{value}_{rtr} = \frac{\text{jobtime}_{avg}}{\text{MRT}}
\]  

(7.2)

The utilization is computed as the ratio between the average number of requests arrived to the microservice in a minute and the MRPM defined in the µService-Descriptor. The Equation 7.3 is used for the computation of the value of the utilization, where \(\text{rpm}_{avg}\) is the average requests per minute for all the known instances of the microservice.

\[
\text{value}_{util} = \frac{\text{rpm}_{avg}}{\text{MRPM}}
\]  

(7.3)
7.3. Evaluation

Policies

The three available policies, i.e. scale-in, scale-out and switch, are all enabled. The scale-in policy has a threshold of 0.35 and takes into consideration only the utilization analytic for the computation of the weight.

The scale-out policy has a threshold of 0.75 and the analytics used for the computation of the weight are both the utilization and the response-time-ratio.

The switch policy has a delta of 0.6 and uses both utilization and response-time-ratio analytics for the computation of the weight.

The choice to use only the utilization analytic for the scale-in weight computation is based on the consideration that the demand of the microservices has been manually set and the MRT is imposed according to this value as two to three times the demand. This would keep the response-time-ratio over the scale-in threshold, reducing the probability of scale-in even if the microservices is underused.

7.3 Evaluation

We wanted to evaluate two different aspects of the system: the reactivity to a sudden increase in the workload (Test I), and the capability of the system to self-adapt and follow a realistic workload (Test II). This second workload is the same bimodal-shaped workload used in Chapter 4 to evaluate the SELFLET Framework. We decided to use the same workload adopted for the evaluation of the SELFLETS to be able to make a direct comparison between the two systems (see Section 7.4). Both the workloads range from 0.1 to 0.8 requests per seconds (RPS). The maximum value of 0.8 RPS has been chosen empirically according to the available resources. Despite the load on the application may appear low, it is worth noting that the scalevideo microservice composing the case study application is a computation intensive task, with an expected demand of almost 30 seconds. We collected the data of 10 runs for each test and we present the mean of the results obtained in all the runs.

In this section we present the results of both the tests that will be followed by a discussion in the next section.

7.3.1 Test I: reactive test

The first test has been studied in order to verify if GRU is able to adapt the application to a sudden increase in the workload. The traffic sent to the application is depicted in Figure 7.2a. We start sending the application 0.1
Chapter 7. Evaluation of the GRU prototype

requests per second (RPS), then after a time interval the RPS are doubled; this happens for three times, reaching a maximum of 0.8 RPS. After every step, the RPS are kept stable for a time interval in order to let the system stabilize. The load last for 3 hours and 45 minutes.

![Workload reactive](image1)

![Workload bimodal](image2)

**Figure 7.2:** Requests sent to the application over time.

The number of active instances of the microservices is depicted in Figure 7.3a. After every step in the RPS, the number of active instances of the microservices is incremented to handle the new workload. There is an initial over-scaling after every step that is due to fact that the requests in the...
7.3. Evaluation

queue should be completed, so the response time of the microservice is over the threshold for an adaptation period. Once the adaptation period is finished, the number of instances are decreased and it is stabilized in order to keep the utilization in the correct range. The scalevideo microservice is the main target of the scaling-out, being the most demanding microservice of the application. Other services are scaled only for few periods to handle a sudden increase of their response time (OBJ2).

![Active Instances](image-url)

(a) Workload reactive

![Active Instances](image-url)

(b) Workload bimodal

**Figure 7.3:** Active instances over time.

The response time of the application (the average every 5 minutes) is
Chapter 7. Evaluation of the GRU prototype

depicted in Figure 7.4a. When there is a step in the RPS the response time goes over the MRP but quickly returns in the desired range of values. This is the effect of the adaptation that scales the number of instances to handle the changes in the RPS (OBJ1).

![Graph](image)

(a) Workload reactive

![Graph](image)

(b) Workload bimodal

**Figure 7.4:** Response time of the application.

7.3.2 Test II: bimodal test

The second test is based on a workload extracted from the data obtained monitoring the traffic of a real website for several days. The original work-
7.4. Discussion

Load has been shrunk from 48 to 6 hours and the number of requests has been scaled to have a peak of 0.8 RPS. The scaling in the number of RPS has been done to adapt the workload to the resources available for the test. The resulting workload presents a bimodal shape and is depicted in Figure 7.2b. GRU should be able to scale the number of active instances of the microservices in order to follow the traffic shape and to keep the response time of the application under its MRP.

The number of active instance for every microservice is depicted in Figure 7.3b. The application is adapted by GRU scaling the microservices to follow the traffic shape (OBJ2). Since the system is reactive, there is an adaptation time needed by GRU to understand the change in the workload and to actuate the needed adaptation actions. However, the response time of the application, depicted in Figure 7.4b, shows that GRU can adapt effectively the application to the workload, ensuring only a few violations of the MRT (OBJ1).

7.4 Discussion

In this section we want to discuss and further analyze the results obtained in the tests. We also want to compare GRU with the SELFLET Framework, discussing the results obtained by the two systems in handling a realistic workload (OBJ4).

7.4.1 Analysis of GRU results

The results of the tests show that GRU can manage an application developed as microservices deployed in Docker containers. GRU can make the application self-adaptive though the interaction with the containers, actuating self-adaptive actions on the basis of a partial knowledge to adapt the application to the variation of the environment where it is running. The self-adaptive actions are still limited to the scaling of the microservices, however this is enough to ensure that the application is adapted to the changing workload in order to respect the constraints imposed by the user (i.e., the MRT). The results show also that, even if the application is not monitored by GRU in its totality, it is sufficient to monitor and respect the constraints imposed on the single microservices to obtain the effective adaptation of the entire application.

Despite the good results obtained in the tests, we want to point out some aspect of the system that are worth noting and that can be improved in the future.
Chapter 7. Evaluation of the GrU prototype

Probabilistic policy selection

We have executed a test of the system using a random policy selection as a baseline case to evaluate the effectiveness of the probabilistic approach.

The initial setting is the same as described in Section 7.2. The selection of the policy to execute at every iteration of the adaptation loop is done randomly, without taking into account the weight of the policies, which is in the range \([0, 1]\). The set of policies considered for the selection comprehend only the valid one for each policy. In particular:

- for the scale-in policy are discarded the policies that violate the constraint on the base-services of the node, and the ones related to microservices that are not running;
- for the scale-out policy are discarded the policies related to services that have not enough resources to be started;
- for the swap policy are discarded the policies that violate the constraint on the base-services of the node, the policies related to services that have not enough resources to be started, and the policies related to services that cannot be compared.

The response time of the application using the random policy selection is depicted in Figure 7.5, while the active instances during time are depicted in Figure 7.6.

Services are scaled without following the changes in the workload, and the system cannot reach a stable state. As a consequence, the response time quickly goes over the maximum one imposed as a constraint. We can conclude that the random selection of a valid policy cannot guarantee the respect of the constraints imposed on the application, and the probabilistic approach outperforms this baseline case.

Scaling Thresholds

We set the thresholds for the scale-in and scale-out policies to 0.35, and 0.75 respectively. These values are consistent with industrial standards. However, we want to compare the results obtained using the same thresholds set for the SELFLETS, i.e., 0.4 for the scale-in policy, and 0.6 for the scale-out policy. We collected the data of 3 tests for each workload and present the mean of the runs.

The response time of the application is slightly improved with both the workloads (see Figure 7.7). This is due to the lower scale-out threshold, that leads to the start of more instances of the services (see Figure 7.8).
7.4. Discussion

(a) Workload reactive

(b) Workload bimodal

Figure 7.5: Response time of the application with random policy selection.
Chapter 7. Evaluation of the GRU prototype

Figure 7.6: Active instances over time with random policy selection.
7.4. Discussion

(a) Workload reactive

(b) Workload bimodal

**Figure 7.7:** Response time of the application using SELFLETS scaling thresholds.
Chapter 7. Evaluation of the GRU prototype

Figure 7.8: Active instances over time using SELFLETS scaling thresholds.
7.4. Discussion

However, the reduced interval between the scale-out and scale-in thresholds may increase the instability of the system, resulting in violations of the maximum response time when the traffic is stable, especially with the reactive workload depicted in Figure 7.7a. A mechanism to dynamically set the thresholds of the policies could improve the performance of the system, and can be the subject of further investigation.

Adaptation loop time interval

In the tests the time interval for the adaptation loop has been set to 120 seconds. This value has been chosen according to the time needed by the microservices to accomplish their task. In particular, the scaleVideo service Max Response Time has been taken as a reference, since it is the microservice that requires more time to complete a request. However, it is important to understand how the variation of the time interval for the adaptation loop can influence the behavior of the system.

Figure 7.9 and Figure 7.10 show the response time and number of instances of the system with a time interval for the adaptation loop set to 90 seconds instead of 120 seconds.

The charts represent the average of five runs. We can see that the system can respond quickly to the changes in the workload, scaling the instances faster to follow the traffic coming to the application. This generally improves also the response time of the application. The drawback is the introduction of a little instability in the system, that may prevent it ensuring the response time under the defined threshold when the traffic stabilizes after a step.

This highlights the importance of the time interval for the adaptation loop, that requires a deeper study and new techniques to tune it the best way.

Resource consumption

GRU is able to manage the resources in a single node, allocating the free amount of resources needed by the microservices. However, it would be good to have a more sophisticated system for the resource allocation.

The CPU usage of the microservices and the cluster is depicted in Figure 7.11 for both the tests. The CPU usage of the microservices is kept more or less constant for the entire duration of the tests, without being affected by the increase or decrease in the number of active instances (OBJ2).

However, the charts show that the cluster results underused for some time intervals. This is due in part to the design of the application itself: the
Chapter 7. Evaluation of the GRU prototype

Figure 7.9: Response Time of the application with a loop time interval of 90 seconds.
7.4. Discussion

(a) Workload reactive

(b) Workload bimodal

**Figure 7.10:** Active instances over time with a loop time interval of 90 seconds.
Chapter 7. Evaluation of the GRU prototype

Figure 7.11: CPU usage of the microservices.
7.4. Discussion

scalevideo microservice has a demand that is several times higher than the other services. The consequence is that the workload has been created taking into consideration mainly that microservice, while the others result underused with the same workload. Despite this consideration, the use of a system based on affinity between microservices may lead to a better usage of the resources of a node, favoring the scaling-out of microservices on the same node that can better use all of its resources. This can be implemented as a future work exploiting the information about the type of the microservices contained in the $\mu$Service-Descriptors. Moreover GRU lacks a system to manage effectively containers requiring heterogeneous resources (e.g., CPUs, memory, etc.). It would be interesting to implement an algorithm to decide which kind of containers create in order to maximize the allocation of resources of the node.

Design of analytics

The design of the analytics of the system is fundamental to obtain an effective adaptation. The values of the analytics used during the evaluation – i.e., response time ratio and utilization – are depicted in Figure 7.12 and Figure 7.13.

The response time ratio follows the variation in the performance of the application and can trigger the scaling of the system to adapt to an increase in the workload for every microservice. We can conclude that this analytic represent a good choice for the adaptation of the application (OBJ3). The utilization analytic is effective with the scaleVideo microservice, but other microservices presents very low values for the entire duration of both the tests. This is in part due to the big difference between the demand of the scaleVideo and the one of the other microservices composing the application. However, despite the system scaled correctly the application also reducing the number of instances when the workload decreased, the analytic could be improved to provide better performance (OBJ3).

Further studies on the design and implementation of analytics can help in the definition of the most effective ones, and can improve the decision making process of the system.

Reactive vs. proactive

GRU is a reactive system, that actuates the self-adaptive actions when certain conditions are met. The results of the tests prove that this can be enough to adapt the application, however the use a proactive system can lead to even better results. This could be implemented as a component between
Chapter 7. Evaluation of the GRU prototype

Figure 7.12: Analytic response time ratio values.

(a) Reactive response time ratio

(b) Bimodal response time ratio
7.4. Discussion

(a) Reactive utilization

(b) Bimodal utilization

Figure 7.13: Analytic utilization values.
Chapter 7. Evaluation of the GRU prototype

the Analyzer and the Planner, that exploits a *Machine Learning* algorithm based on the partial view of the system built by the Analyzer to predict the adaptation needed.

**Self-adaptive capabilities**

GRU implements only policies that involve the scaling of the containers running the microservices. This is enough to adapt the application to the variation in the workload. However, the implementation of new policies based on different actions can lead to more interesting and sophisticated self-adaptive capabilities, such as the migration or re-configuration of microservices. The policies and strategies creation could be left to the user, that can customize the self-adaptive actions and the decision making process according to its needs.

The purpose of the tests was to demonstrate the application of GRU to an application that is not explicitly design to be integrated with it, so the interaction between GRU and the application is transparent and happens exploiting the containers. The creation of an application that is designed to be integrated with GRU, maybe exposing in every microservice an endpoint to receive some commands by GRU itself, can further improve the self-adaptive capabilities of the system.

**Different adaptation levels**

GRU actuates self-adaptive actions only on the containers running the microservices of the application. It would be interesting to make GRU able to operate at different levels of adaptations that may involve the topology of the application or the automatic re-configuration of the GRU-Agent themselves.

7.4.2 *Comparison with the SELFLET Framework*

Both GRU and the SELFLET Framework have been tested using the bimodal-shaped workload to validate their capability to self-adapt the application to a realistic workload. It should be noted that the SELFLET Framework presents more sophisticated self-adaptive feature, having been developed for a longer time; GRU is still an initial prototype and presents the limitations described in the previous section. However, a comparison on the performance of the two systems can be useful to understand the possible improvements and the issues that should be addressed in GRU (OBJ4).

We can compare the response time of the application, depicted in Figure 7.4b and Figure 4.6b. GRU keeps the response time of the application
under the threshold for almost all the time, presenting only few violations of the constraint: The SELFLET Framework provides slightly better performance, keeping the response time of the application always under the threshold.

The two systems handle the resources differently. GRU can scale the instances of every single microservices, while the SELFLET Framework scales the number of nodes running the SELFLETs that will run several services. Since every microservice has allocated one core of the CPU and the maximum number of active instances of all the microservices is 50 (the peak in Figure 7.3b), the maximum number of nodes allocated by GRU can be approximated to 25 nodes. The SELFLETs allocate a maximum number of 39 nodes using the dynamic scaling (Figure 4.7b). The use of containers running a single service provides a fine grained and more effective scaling, reducing the number of active instances and avoiding to waste resources. The SELFLETs present a better use of the CPU of the system, that is kept more or less constant during the test, as depicted in Figure 4.8b. GRU has not the same performance, as described in the previous section and depicted in Figure 7.11b. Both SELFLETs and GRU use a partial view of the system for the decision making process, but the SELFLETs are organized in Neighborhoods by the Requests Dispatcher, while GRU-Agents choose the peers to communicate with randomly. The approach adopted by GRU is less computationally expensive, but it does not affect significantly the performance of the application.

We can conclude that the SELFLET Framework can guarantee the satisfaction of the constraint imposed on the Maximum Response Time of the application, while GRU presents a small number of violations but allocates almost half of the resources of the SELFLETs.

It is worth noting that GRU can be greatly improved addressing some issues that are related to the early stage of development of the project. The use of an external approach and the Docker containers have proven to be a valid choice for the transparent integration of Self-Adaptation inside a distributed system. The advantage of GRU is its easy integration with a wide range of applications in a transparent way. Its self-adaptive capabilities can be improved with further studies, and can be extended allowing the application to communicate directly with the GRU-Agents. In this way, the user can choose between a totally transparent approach (the best solution for a system already developed), and a more integrated and effective approach.
Chapter 7. Evaluation of the GRU prototype

7.5 Conclusion

In this chapter we presented our evaluation of the GRU tool. This evaluation is important to validate our external approach to introduce transparent Self-Adaptation in a distributed system. We described how we implemented the Video-Provisioner use case as a microservices application running in Docker containers.

We evaluated GRU in two different tests to verify its reactivity in the adaptation process and its capability to handle a realistic workload. We presented the results of such evaluation that demonstrate the validity of our approach. GRU is able to make self-adaptive an external distributed system in a transparent way, managing the application in order to keep its response time under the user defined threshold with only few violations. The fine grained scaling can avoid the waste of resources, minimizing the over- or under-scaling of the microservices instances.

The discussion about the limitations and possible improvements of GRU, as well as its comparison with the SELFLET Framework, is fundamental to understand the topics that should be the focus of future works. The limitations of GRU are mostly related to the management of resources (such as the CPU utilization) and the limited self-adaptive capabilities it can provide at this stage of development. However, this issues will be addressed in future work on the system, which we can conclude obtained promising results in our evaluation.
CHAPTER 8

Conclusion

This thesis investigated the design and management of distributed Self-Adaptive Systems, in particular the ones adopting a decentralized approach for the self-adaptive process. The focus of our work was not on algorithms for such kind of systems, but on the design of software engineering approaches to effectively apply Self-Adaptation to distributed systems.

The contributions of our work are well described in Chapter 1 and can be summarized in three major points:

- The robust evaluation and analysis of a decentralized Self-Adaptive System – i.e., the SELFLET Framework – through a realistic case study to understand the effectiveness and utility of decentralized Self-Adaptation in a distributed system;

- The design of a tool, derived from the previous analysis, to enable transparent adaptation in Complex Systems;

- The implementation and evaluation of a prototype of such tool – i.e., GRU – to validate the proposed design.

The SELFLET Framework has been a good starting point for our research. It was an existing system that we improved by implementing new
Chapter 8. Conclusion

algorithms and working on the fine tuning of its parameters to obtain the best performance.

The development of a realistic case study derived from the video-on-demand scenario enabled a robust evaluation of its capabilities. Through this evaluation we proved the validity of the theoretical concepts behind the SELFLETS, such as the decentralization, the emergent behavior and the encapsulation of the business logic inside containers.

The results of the evaluation of the SELFLET Framework are the basis of our critical analysis of its design choices. We described every aspect pointing out advantages and limitations. The main issues that have been identified for the concrete adoption of the SELFLETS have been related to the use of the framework: despite it could provide several advantages in the development process, it requires the developers to learn how to use it, and impose constraints on the technology of the application. Some improvements can be done also in the use of the containers for the encapsulation of the business logic.

These considerations led us to the design of a tool that inherits all the advantages of the SELFLET Framework but tries to overcome its limitations. The separation of the adaptation manager and the managed element, and the use of the adaptation enabler to connect them is the core of the proposed design. We also identified the technologies best suited for the concrete implementation of these parts: the Microservices Architecture pattern is a perfect match for the managed element, while Docker containers can be exploited as the adaptation enabler, being the de facto standard in virtual container technology. The adaptation manager is a Multi-Agent System that follows the design of the SELFLETS.

In order to validate the design of our tool, we implemented a prototype called GRU that can bring self-adaptive capabilities in a transparent way to a distributed system built using the Microservices Architecture pattern and deployed in Docker containers. We described GRU, providing the details regarding its architecture and its implementation. The main advantage of GRU is its easy and transparent integration in a possible wide range of applications, as well as its high customization.

The evaluation of GRU has been done using the same case study previously defined for the evaluation of the SELFLET Framework, but imple-
mented as an application based on the Microservices Architecture pattern. The results of the evaluation validate the design of GRU as a way to easily integrate Self-Adaptation into an external distributed system. We also discussed such results comparing GRU and the SELFLETS, highlighting the limitations of the current prototype that could be address in future works.
CHAPTER 9

Future Work

The last part of our research has been dedicated to the design of the tool to enable self-adaptive capabilities in distributed systems in a transparent way, and the implementation of its prototype called GRU. The evaluation we did on the prototype provided good results validating both the design of the tool and GRU. However, it highlighted also some issues and limitations, that could be addressed in future work.

In this chapter we want to focus on the future work related to the design of tool and the GRU prototype, discussing the possible improvements that can be the topic of further investigation.

9.1 Future work related to the design of the tool

The design of the proposed tool can benefit from further evaluations and analysis.

We applied the tool to the domain of the applications developed with the Microservices Architecture pattern and deployed in a cloud computing environment. It could be interesting the application of the tool to a different domain (e.g., Cyber-Physical Systems, Internet of Things, etc.) to understand its flexibility to different scenarios.
Chapter 9. Future Work

The adaptation enabler is an interesting topic to explore. We adopted the Docker containers for their maturity and diffusion in industry. The interaction between Docker and the adaptation manager is still limited, so improvements in this direction can have an impact on the effectiveness of the adaptation process.

Docker containers are not the only possible solution. There are other implementations of the virtual containers technology, like the Rocket containers [5], that may provide some advantages and that are worth investigating.

9.2 Future work related to the GRU prototype

GRU is still an early prototype and can benefit from several improvements, as pointed out in our discussion in Chapter [7].

The self-adaptive capabilities provided by GRU are limited to the scaling of the application. It is necessary to expand them to create a complete self-adaptive solution. The implementation of new policies and actions can enable a more sophisticated Self-Adaptation, making GRU able to autonomously balance the load between services, handle failures at different levels, migrate services to change the topology of the application and improve the resource consumption. The creation of policies for these operations is not trivial and requires the study of algorithms for the effective computation of the weights of such policies.

The time for the adaptation loop requires a deeper study. It would be interesting to implement a dynamic tuning of this parameter, which may depend on the characteristics of the microservices running in the node. This way the system can self-configure to provide the best performance with every application.

The decision making process is now based on a very simple probabilistic algorithm implemented as a strategy inside the system. The development of new strategies based on more sophisticated algorithms can greatly improve the quality of the decision taken by the GRU-Agents about the policy to actuate.

GRU does not integrate a proactive adaptation process. The integration of a predictor inside the adaptation loop (e.g., between the Analyzer and the Planner) can lead to the creation of proactive policies able to anticipate future changes. This requires the study of Machine Learning approaches that can predict such changes starting from the data provided by the Analyzer component.

The usage of resources could be improved. The study of new algorithms
9.2. Future work related to the GRU prototype

for the optimization of containers placement can greatly improve the usage of resources inside a node of the cluster. A placement based on the idea of “affinity” between containers could be investigated: from the description of the resources needed by a microservices and provided by the user in the μService-Descriptors, the system can try to understand the best composition of containers inside a node to better use its resources.

The customization of the system is still limited. The development of a system to allow the user to create its own policies and strategies that can be read by GRU is needed to enable the effective application of the tool to a wide range of different applications.


Bibliography


Bibliography


Bibliography

Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


