POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

SELF-ADAPTABILITY VIA HEARTBEATS AND CENTRAL MULTI-APPLICATION COORDINATION

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Ing. Martina MAGGIO

Correlatore: Prof. Giovanni SQUILLERO

Tesi di Laurea di:

Marco TRIVERIO

Matricola n. 735184

Anno Accademico 2009–2010

To my family. To Lorenza.

Acknowledgments

There are many people who have made this thesis possible and that, in some way or another, have supported me during these years. I want to thank them because I am truly indebted to them.

To my parents, Claudia and Riccardo. You have given me everything I ever needed.

To my sister, Silvia. You are by my side when no one else is, thank you.

To my fiancée, Lorenza. I want to build something great with you.

To my advisor, Marco. It has been a pleasure to work with you.

To the research group I am part of, Filippo and Luca in particular.

To all of my friends: the ones in Biella, the ones in Milano, and the ones in Chicago.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AMD	Advanced Micro Devices
API	Application Programming Interface
ABI	Application Binary Interface
BORPH	Berkeley Os for ReProgrammable Hardware
CBSE	Component-Based Software Engineering
CHANGE	Computing In Heterogeneous Autonomous aNd Goal-oriented Environments
CSAIL	Computer Science and Artificial Intelligence Laboratory
CO	Clustered Object
COID	Clustered Object Identifier
COR	Clustered Object Reference
COS	Clustered Objects System
CPU	Central Processing Unit
DES	Data Encryption Standard
DDR	Double Data Rate
DLL	Dynamic-Link Library

DVS	Dynamic Voltage Scaling
EC2	Elastic Compute Cloud
FAT	File Allocation Table
fos	Factored Operating System
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GNU	GNU is Not UNIX
GPGPU	General Purpose Graphics Processing Unit
GPL	General Public License
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GTT	Global Translation Table
HDL	Hardware Description Language
HPC	High-Performance Computing
IBM	International Business Machines
ICAP	Internal Configuration Access Port
I/O	Input/Output
IPC	Inter-Process Communication
IPCM	IP-Core Manager
IP-Core	Intellectual Property Core
ISS	Implementation Switch Service

JIT	Just In Time
KFS	K42 File System
KST	Kernel Symbol Table
LKM	Loadable kernel module
LTT	Linux Trace Toolkit
LTT	Local Translation Table
MAC	Media Access Controller
MAPE-K	Monitor, Analyze, Plan, and Execute with Knowledge
MDP	Markov Decision Process
MIMD	Multiple Instruction Multiple Data
MIPS	Microprocessor without Interlocked Pipeline Stages
MIT	Massachusetts Institute of Technology
MPMC	Multi-Port Memory Controller
NFS	Network File System
NUMA	Non-Uniform Memory Access
ODA	Observe–Decide–Act
OTL	Organic Template Library
OTT	Object Translation Table
PA	Performance Assertions
PCI	Peripheral Component Interconnect
PDR	Partial Dynamic Reconfiguration

PCS	Performance Counter Subsystem
PID	Process Identifier
PLB	Processor Local Bus
QoS	Quality-of-Service
RCU	Read Copy Update
RISC	Reduced Instruction Set Computer
RSoC	Reconfigurable System-on-Chip
SDRAM	Synchronous Dynamic Random Access Memory
SMMP	Shared-Memory symmetric Multi-Processor
SOA	Service-Oriented Architecture
SoC	System-on-Chip
SPMD	Single Program Multiple Data
STAPL	Standard Template Adaptive Parallel Library
STL	Standard Template Library
TID	Thread Identifier
UART	Universal Asynchronous Receiver-Transmitter
UML	Unified Modeling Language
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XUPV2P	Xilinx University Program Virtex-II Pro

Sommario

La complessità dei sistemi informatici è aumentata esponenzialmente negli ultimi decenni. Ingegneri del software e programmatori fronteggiano ogni giorno sistemi che richiedono parecchio tempo e notevoli conoscenze tecniche per essere ottimizzati al fine di offrire prestazioni ottimali. È ormai evidente che non è più possibile contare esclusivamente sull'intervento umano per regolare ed ottimizare un sistema: le condizioni di esecuzione cambiano costantemente, rapidamente e in maniera imprevedibile. Per far fronte a questi nuovi scenari è necessario che i sistemi siano in grado di *adattarsi* automaticamente alle mutazioni dell'ambiente in cui sono inseriti. Questa tesi analizza questa problematica di ricerca partendo da un approccio radicalmente nuovo e ponendo le basi per un framework che permetta a sistemi software ed hardware di autoregolarsi a runtime.

L'architettura qui presentata estende gli attuali sistemi tramite l'impiego del loop Observe–Decide–Act (ODA); questo meccanismo fornisce la capacità di *osservare* stato e performance di un dato sistema, di *decidere* quali modifiche pianificare al fine di ottimizzare determinate operazioni e di *attuare* dette modifiche. L'analisi dello stato e delle performance del sistema avviene attraverso l'uso di specifici componenti detti *sensori*, che raccolgono numerose informazioni relative al sistema; i cambiamenti vengono invece concretizzati da moduli generalmente chiamati *attuatori* (o *servizi*); infine, le decisioni vengono prese da uno specifico elemento chiamato *decision engine*, che raccoglie tutte l'esperienza sensoriale e governa gli at-

tuatori attraverso criteri che possono essere o semplicemente euristici o guidati da un sofisticato agente di apprendimento a rinforzo (*reinforcement learning*) in grado di adattarsi in maniera dinamica all'ambiente. L'utilizzo dell'approccio ODA rappresenta uno dei primi passi verso la realizzazione di sistemi in grado di reagire a situazioni sfavorevoli, imprevedibili e/o sconosciute. Questi sistemi sono detti *self-adaptive* e fanno parte del cambiamento che sta portando verso sistemi *goal-orientated*; una della principali proprietà di tali sistemi è che questi conoscono gli obiettivi da raggiungere (ovvero, il *cosa*) ma potrebbero non sapere *come* raggiungerli o potrebbero adattare la propria esecuzione in base al contesto specifico.
Questo lavoro fa parte del progetto di ricerca Computing In Heterogeneous Autonomous aNd Goal-oriented Environments (CHANGE), intrapreso dal Laboratorio di Micro-Architetture del Politecnico di Milano in collaborazione con il Computer Science and Artificial Intelligence Laboratory (CSAIL) del Massachusetts Institute of Technology (MIT), e si propone di presentare l'architettura necessaria per costruire questo di sistemi self-adaptive. Il punto di partenza è un componente di *monitoring* delle performance sviluppato presso il CSAIL del MIT[1] e noto come *Application Heartbeats API* [3, 4]. Gli *Application Heartbeats* sono un semplice, leggero ma al tempo stesso potente meccanismo in grado di misurare la performance rispetto a precisi obiettivi. Sulla base di questo particolare tipo di *sensori*, abbiamo sviluppato un *decision engine* (chiamato *Consensus Object*), degli attuatori, e un'infrastruttura

[1] Questa ricerca ha anche collegamenti con altri progetti del CSAIL del MIT come il Factored Operating System (fos), gli *Smartlocks* [1] e le tecniche di perforazione del codice [2]; questi verranno descritti con maggiore dettaglio nel Capitolo 2 e nel Capitolo 3. La tesi renderà esplicito quali parti sono stati realizzati dal gruppo del *Massachusetts Institute of Technology* e quali sono invece frutto del mio lavoro individuale. In particolare userò i termini 'we", "our", "I" e "my" in riferimento al lavoro che ho realizzato sotto la supervisione del mio Relatore, il Professor Santambrogio. Attribuirò esplicitamente il lavoro svolto dal gruppo di ricerca del CSAIL del MIT

comunicativa per collegare questi componenti. In particolare abbiamo progettato e implementato: un'entità decisionale centralizzata, che raccoglie tutte le informazioni riguardanti le performance di sistema e che è in grado di decidere quali misure analizzare; i canali comunicativi attraverso i quali le decisioni sono trasmesse; diversi elementi che provocano cambiamenti nell'ambiente. Molta attenzione è stata dedicata all'algoritmo decisionale che gestisce il comportamento del *Consensus Object*; in particolare, in seguito all'analisi di varie tecniche di machine learning, abbiamo deciso di utilizzare l'apprendimento a rinforzo come base per il nostro decision engine. Questo metodo non calcola a priori un modello dell'ambiente ma interagisce con il sistema ottenendo rinforzi quando determinate condizioni vengano verificate, ad esempio quando tutti gli obiettivi di performance sono stati raggiunti; l'algoritmo impara dunque quali attuatori attivare sulla base delle ricompense precedenti. Inoltre tutti i canali di comunicazione tra elementi del sistema sono stati profondamente ottimizzati al fine di ridurre l'overhead dell'intera infrastruttura.

Il contributo innovativo di questa tesi sta nel proporre un framework per costruire sistemi adattivi che è:

- completo, perché implementa i concetti dell'ODA (monitoraggio, decisione e attuazione)
- semplice ma al tempo stesso utilizzabile in scenari complessi
- leggero, come dimostrato nel Capitolo 5, poiché presenta basso overheard sia riguardo le applicazioni monitorate sia riguardo gli attuatori gestiti
- flessibile, grazie alle funzionalità di di machine learning.

Queste caratteristiche, unite a risultati sperimentali incoraggianti, dimostrano che la soluzione proposta ha il potenziale per passare da *enabling technology* a vero e proprio prodotto funzionante.

La tesi è organizzata come segue. Nel **Capitolo 1** viene descritto il contesto di ricerca, viene fatta una panoramica delle discipline coinvolte e vengono spiegati i concetti fondamentali dei sistemi adattivi. Il **Capitolo 2** analizza le ricerche relative all'argomento e descrive lo Stato dell'Arte dei sistemi self-adaptive. Il loop ODA, insieme a tutte le sue componenti (sensori, attuatori e decision angine), vengono spiegato in dettaglio. Vengono inoltre trattati i diversi approcci all'inserimento dei sensori nelle applicazioni (supported e forced instrumentation). Il **Capitolo 3** descrive nel dettaglio la soluzione proposta. Partendo dalla relazione con il loop ODA delineiamo i tratti principali del decision angine (*Consensus Object*) ed infine spieghiamo i suoi canali di comunicazione e le sue capacità di machine learning. Inoltre le interfacce con i sensori e con gli attuatori (quest'ultima è chiamata *Services API*) vengono descritte e giustificate in dettaglio. Nel **Capitolo 4** vengono presentate le implementazioni: il *Consensus Object*, il suo algoritmo di apprendimento a rinforzo, la *Services API* e una libreria self-adaptive che realizza le funzionalità del Data Encryption Standard (DES). Sono inoltre elencate le linee guida per gli sviluppatori che volessero estendere l'implementazione. I risultati dei test che valutano a livello sperimentale l'implementazione del *Consensus Object*, della *Services API* e della libreria self-adaptive sono invece presentate nel **Capitolo 5**. Qui vengono quantificati gli overhead dell'implementazione dell'architettura self-adaptive. Il **Capitolo 6** conclude la tesi con alcune considerazioni relative al lavoro svolto e a possibili sviluppi futuri.

Abstract

Nowadays the complexity of computing systems is skyrocketing. Programmers have to deal with extremely powerful computing systems that take time and considerable skills to be instructed to perform at their best. It is clear that it is not feasible to rely on human intervention to tune a system: conditions change constantly, rapidly, and unpredictably. It would be desirable to have the system automatically *adapt* to the mutating environment. This dissertation analyzes the stated problem, embraces a radically new approach, and lays the foundations of a framework that enables software and hardware systems to adjust during execution.

The architecture presented in this paper augments current computing systems through the Observe–Decide–Act (ODA) loop mechanisms, which provide the capability of *observing* the status and the performance of a given system, of *deciding* which modifications to undertake in order to optimize certain operations, and of *enacting* such modifications. Observation is carried out through the use of specific components called *sensors*, which gather various types of information about the system; *actions* are undertaken by modules usually called *actuators* (or *services*); decisions are taken by a precise element called *decision engine*, which gathers all sensorial experience and steers actuators through a policy that might be a simple heuristic or a more sophisticated reinforcement learning agent capable of dynamically adapting to the environment. The adoption of the ODA approach stands as one of the first steps towards making it possible to build systems capable of

reacting to unfavorable, unpredictable, and unknown situations. Such systems are called *self-adaptive* or *autonomic* systems and they are part of the paradigm shift towards *goal-orientation*; one of their property in fact is that they know their objectives (the *what*) but they might not know the way (the *how*) to reach them or they might adapt their execution path depending on the context.

This work introduces the complete architecture to build such types of systems in the context given by the Computing In Heterogeneous Autonomous aNd Goal-oriented Environments (CHANGE) project, a research initiative of the Laboratory of Micro-architectures at the Politecnico di Milano in collaboration with the Massachusetts Institute of Technology (MIT) Computer Science and Artificial Intelligence Laboratory (CSAIL). The starting point is an open-source performance *monitoring* facility developed at the MIT CSAIL[2] called *Application Heartbeats API* [3, 4]. The *Application Heartbeats* are a simple, lightweight, yet powerful mechanism to supervise the performance of given targets. Building on the top of that particular kind of *sensors*, we have developed a *decision engine* (called *Consensus Object*), actuators, and the communication infrastructure to link such components. In particular we have designed and implemented: a centralized decision entity, which gathers all the information about system performance and which is able to decide which measures to enact; the communication channels through which the decisions are conveyed; the elements that commit changes to the environment in different possible ways. Much attention is

[2] My work has also some blander links with other MIT CSAIL project such as the Factored Operating System (fos), *Smartlocks* [1], and *Code Perforation techniques* [2]. All of these will be discussed in more detail in Chapter 2 and Chapter 3. In this dissertation I will explicitly disambiguate the work carried out by the *Massachusetts Institute of Technology* group and my individual work. In particular I will use the terms ”we”, ”our”, ”I”, and ”my” to refer to the work I have done under the supervision of my advisor, Professor Santambrogio. I will explicitly attribute work done by MIT CSAIL group

provided to the decision algorithm that drives the behavior of the *Consensus Object*; in particular, after analyzing several different machine learning techniques, we have chosen to embrace reinforcement learning as the foundation for the decision engine. This approach does not precompute a model of the environment; it interacts with the system obtaining rewards when certain conditions are achieved (*i.e.,* all the performance goals are accomplished) and learning which actuators to enable from past rewards. Moreover all the communication channels between elements of the system have been deeply optimized in order to reduce the overhead of the whole infrastructure.

The novel contribution of this thesis is to provide a framework to build autonomic systems that is:

- complete, since it implements the ODA concepts (monitoring, decision, and enactment)
- simple yet capable of being used in complex scenarios
- lightweight, as demonstrated in Chapter 5, since it has a low overhead both on the monitored applications and on the handled actuators
- flexible, as a result of the machine learning capabilities, in order to be able react to changing conditions.

Such characteristics, along with the encouraging experimental results, prove that the proposed solution has the potential to be extended and to transition from an *enabling technology* to a working product.

The remainder of dissertation is organized as follows. In **Chapter 1** the research context is described and some of the basic concepts of autonomic computing are introduced; moreover, an overview of the involved disciplines is given. **Chapter 2** reviews related researches and describes the

State-of-the-Art in self-adaptive systems. The ODA loop is explained in detail as well as its components: sensors, actuators, and the decision engine. Supported and forced instrumentation are presented to explain the different approaches to inserting sensors in applications. **Chapter 3** describes in detail the proposed solution. In particular we start from the relation with the ODA loop and then delineate the main traits of the decision engine (*Consensus Object*): its machine learning capabilities and its communication channels are explained. In particular, the interface with the sensors and the interface with the actuators (called *Services API*) are traced and justified. In **Chapter 4** the implementations are described: the *Consensus Object*, its reinforcement learning algorithm, the *Services API*, and a self-adaptive library that realizes the Data Encryption Standard (DES) encryption and decryption functionalities. Moreover guidelines for developers that might want to extend the implementation are outlined. The results of the tests that experimentally evaluate the implementations of the *Consensus Object*, the *Services API*, and the self-adaptive library are presented in **Chapter 5**. The overhead of the implemented self-adaptive architecture is quantified. Finally, **Chapter 6** ends this paper with some considerations on the work carried out and proposing potential future works.

Chapter 1

Introduction

"...Bond. James Bond."
Dr. No *(1962)*

This Chapter provides an introduction to the context in which this dissertation is settled and to the basic concepts that will be referred to throughout all the other Chapters.

1.1 Introduction to the problem

Nowadays computer systems are rapidly evolving and are becoming increasingly complex. Complexity is upturning difficulties in keeping a system maintainable. In fact, even before a certain product is launched, some issues, such as bugs and security issues, might already be known. The system has then to be updated in a fast cost-efficient manner. This drastically shortens the life-cycle of products and makes both hardware and software subject to an unprecedented rate of change. There is a strong need to make future computer systems not only easily upgradeable but also able to:

- reduce (ideally to zero) down-time caused by updates. At present down-time is in fact reducing dependability and availability of systems. Some steps in this direction are represented by the work done in K42 [5].
- be proactive and suggestive towards changes. The stimulus for an improvement or optimization should not only be driven by an external entity but might be advocated by the system itself. This is a ferment field of research that will be discussed in Section 2.

The problem has been approached from two different perspectives: *hardware* and *software* adaptability. As we will see, these two approaches are usually considered unrelated. Section 5 will show a novel fusion of the two.

1.1.1 Hardware adaptability

Hardware adaptability has been actively researched since the introduction of Field Programmable Gate Array (FPGA) at the beginning of the 1990's [6]. FPGAs are usually part of hardware-software co-systems where the computationally critical parts (also called *kernels* [7]) are implemented in hardware to boost performance while the rest (usually controller-related tasks) is implemented in software. The most studied and probably most innovative feature of FPGAs is *dynamic reconfigurability* [8, 9]: FPGAs are configured with one or more configuration files called *bitstreams* that describe a certain circuit to be implemented on the board.

While a complete reconfiguration usually interfere with functionality (for this reason the service provided by the FPGA is suspended), it is also possible to only *partially* reconfigure the board. Partial Dynamic Reconfiguration (PDR) enables reconfiguration of a part of the device at runtime without disrupt of functionality [10].

As outlined in [7] there is now a shift from *reconfigurable architectures* to *autonomic systems* capable of configuring, healing, optimizing, and protecting themselves. An instance of FPGA-based online adaptive system has been described in [11] where PDR is used to change the architectural implementation with the aim of optimizing power consumption.
Although interesting, powerful, and extremely fast compared to software solutions, FPGAs have not become as widely adopted as expected. Cases of FPGA-based autonomic computing remain rare and it is contended that radically different architectures will soon be available and predominant: *multi-cores* are already extensively embraced and some expect silicons with *thousands* of cores to make their appearance in the next ten years [12].
FPGAs are not the only examples of adaptable hardware. Current architectures offer several other mechanisms that alter the status of the hardware and, consequently, performance. For instance, *frequency scaling* [13] is a technique that reduces processor clock by some multiple of the maximum [14] with the aim of making the CPU consume less power (at the expense of performance). *Clock throttling* is a technique that does not alter the frequency yet it gates the clock signal for some number of cycles at regular intervals, slowing down the computation. While frequency scaling does not require a change in voltage, Dynamic Voltage Scaling (DVS) [15] lowers power consumption by lowering both the operating voltage and (proportionally) frequency.

1.1.2 Software adaptability

Software is a key element of modern computing systems: it drives the hardware and enables exploitation of its full power. The complexity of software is growing exponentially [16]: programs are often developed by several programmers, built of several inter-communicating components, and run in unknown and mostly unpredictable contexts. Even with modern testing techniques, this scenario often leads to software being deployed with relevant issues. Bugs and security issues have to (or should be) patched in a fast cost-efficient manner through update mechanisms built in the shipped software. But this is only part of the whole problem: in fact, not only software has to be polished and perfected after installation but it has to be designed in order to accommodate needs and fulfill requirements that become evident only time after deployment.

In order to understand this aspect it is useful to review the evolution of software design.

Software engineering was born in the late 1960's when the study of a series of techniques to approach software design, development, and maintenance arose [17]. The goal was desirable: being able to improve the quality of products building them in a more systematic and disciplined way. For the first time a solid alternative to the "code-and-fix" approach (Figure 1.1.a) was proposed: creating software was not synonym for programming anymore. In particular software started to be developed through the following phases (also shown in Figure 1.1.b):

- *Definition of stable requirements*. It starts with the analysis of the context in which the product will be used, the users that will make use of it, and the needs to be fulfilled.

- *Software design*. It is usually driven by tools such as the Unified Modeling Language (UML) and it is used to architect the software before it is implemented: the purpose it to make the software maintainable, modular, expandable, and reusable.

- *Software implementation*. It is the final step in engineering a software and it is based on the given design. It usually consists of programming and testing.

It was indeed argued that a conscientious definition of requirements could avoid radical changes in response to the demand for possibly simple new features: this clearly was a big step towards better quality.

Yet, after just a few years, this approach became not satisfactory because it was based on the arguable assumption that such thing as "stable requirements" existed [18]. When software products became something more than static, monolithic, and centralized tools internal to organizations the so-called *closed world assumption* [18] did not hold anymore. Evolution is indeed intrinsic to software [19] and for this reason software engineers must design programs that can easily accommodate future changes. As outlined by [20] programs must be *designed for change*.

If in the previous years one single change could have implied recompilation and redeployment of the whole system, changes gradually became easier to introduce, also thanks to *object-oriented* programming languages. Software became modular, dynamic and distributed (also across networks). This has been shown in Figure 1.1.c.

Yet the growth in the dynamism of change has not shown signs of stopping any time soon. Even though most softwares developed in the past few years are easily updatable and have little downtime, new application and new domains demand more flexibility and performance. For instance, VisaNet, a network that handles credit card payments, is updated an average of 20000 times a year yet it is always above 99.5% availability [21].

Software still requires a lot of (costly) manual intervention, it sometimes still fails, and it has almost no knowledge about itself: the problem lays in the *open-loop structure* [22] followed in the engineering of the software. There is a wide consensus on the fact that new paradigms have to be explored and new frameworks have to be developed. Among those, *autonomic computing* systems seem to be the response to most of the problems previously described [22]. They introduce a new framework based on the *closed-*

loop approach, which, through a feedback loop, adjusts the software while it is running (Figure 1.1.d). This kind of system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions. The life-cycle of self-adaptive softwares does not stop after its development and initial setup but it indeed continues to handle changes on several possible levels.
The evolution of software engineering as described in this section is delineated in Figure 1.1.

1.2 Autonomic software systems

1.2.1 Terminology

Inspired by the lexicon embraced in [23] and by the willingness of not introducing useless complexity in the adopted vocabulary, in this dissertation I will interchangeably use the terms “autonomic”, “self-adaptive”, “self-managing”, and “organic”.

1.2.2 Motivation

As briefly described in Section 1.1.2, the primary motivation for autonomic computing is rooted in the fact that the costs of handling the complexity of software systems to achieve their goals is drastically increasing [24]. In particular, traditional software has to face great challenges[22] such as:

- complexity of management and maintainance: manual operations are still required to supervise, update, fix,. . . a system.
- changing priorities and policies governing the goals

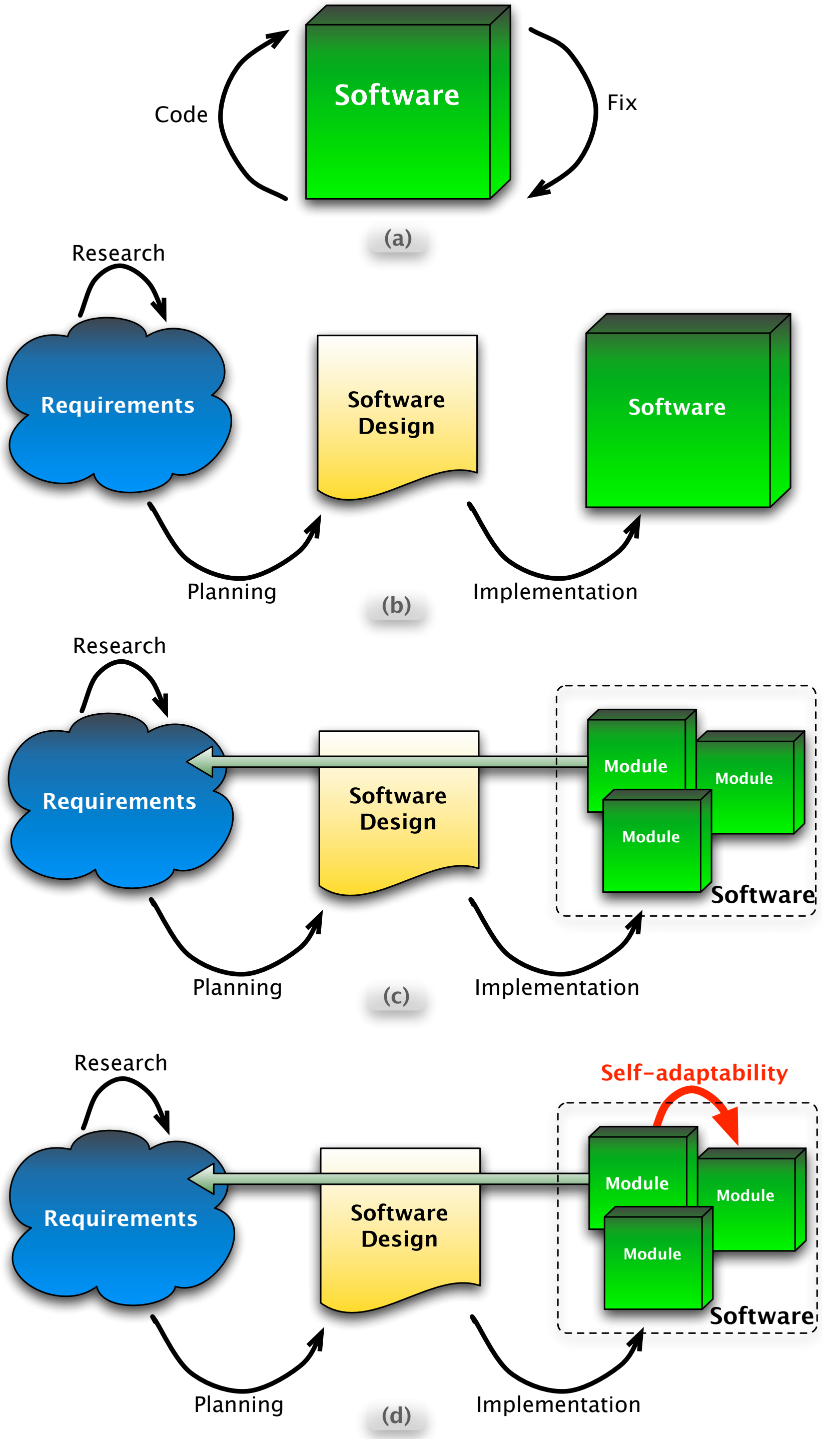


Figure 1.1: The evolution of software engineering

- resistance to unexpected conditions at runtime, which include:
  - hardware failures
  - software failures (such as severe bugs, security breaches, etc, . . .)
  - contexts and conditions that are unknown to the software because they have not been considered in their design and implementation

The long-term goal is not only to build a system that does not have to be stopped in order to be updated but also to make it autonomous and intelligent enough to be able to trigger by itself the best possible changes.

We contend that software self-adaptability is becoming an important topic of research and is gradually earning relevance. As delineated in Chapter 2 there are many instances and contexts where self-adaptability is being investigated and implemented. In particular autonomic computing is becoming increasingly common in large-scale systems. We envision similar characteristics spread also among less comprehensive systems, such as consumer software.

We strongly believe that augmenting software with *organic* attribute will lead to a new era of computing.

1.2.3 Definition and history

Several definition of self-adaptive software have been provided through the years. Among the firsts:

- “Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible” [25]
- “Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean

> anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation” [26]

But autonomic computing was well formalized only in 2001 by IBM with its “Autonomic Computing Manifesto” [16], first presented by Dr. Paul Horn’s during the National Academy of Engineering meeting of the same year. Horn outlined eight key properties (“elements”) of autonomic computing systems:

1. **Self-Awareness or Introspection** — the system must know itself in detail: components, layers, statuses, connections, extents,. . . are necessary parts to be monitored and kept under inspection.

2. **Self-Reconfigurability** — the system must be able to dynamically change its behavior at runtime.

3. **Continuous Self-Optimization** — the system must never settle but it must continuously monitor itself to discover possible optimizations.

4. **Self-Healing Capabilities** — in case of failures (caused by routine or extraordinary events) the system must be able to recover.

5. **Self-Protection Techniques** — the system must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

6. **Context-Awareness** — the system must know the environment (defined as: everything in the operating environment that affects the system’s properties [22]) and the context that surround its activity and take it into consideration when acting.

7. **Openness** — the system must be able to interact with heterogeneous components that might not be under its control.

8. **Anticipation and Support** — the system must be able to provide and anticipate the correct status to handle future changes.

This “flat” description has evolved in the discussion provided by [22], which proposes a hierarchical view of the so-called *self-* properties* (see also Figure 1.2). Three levels are defined:

General level This level contains global properties of self-adaptive software. Self-adaptiveness is a very general term that includes self-managing, self-governing, self-maintenance, self-control, self-evaluating, and self-organizing.

Major level These properties have been defined in accordance to biological self-adaptation mechanisms. For this reason autonomic systems are often said “organic”. An “organism” with such properties has to adapt to the context (just like the human body does, for instance in reaction to a temperature change) or the system itself. Those are the properties of this level in detail, which have been described previously:

- Self-reconfiguring
- Self-healing
- Self-optimizing
- Self-protecting

Primitive level This level features the most basic properties onto which all the others build on.

- Self-awareness
- Context-awareness
- (optional) Openness and anticipation

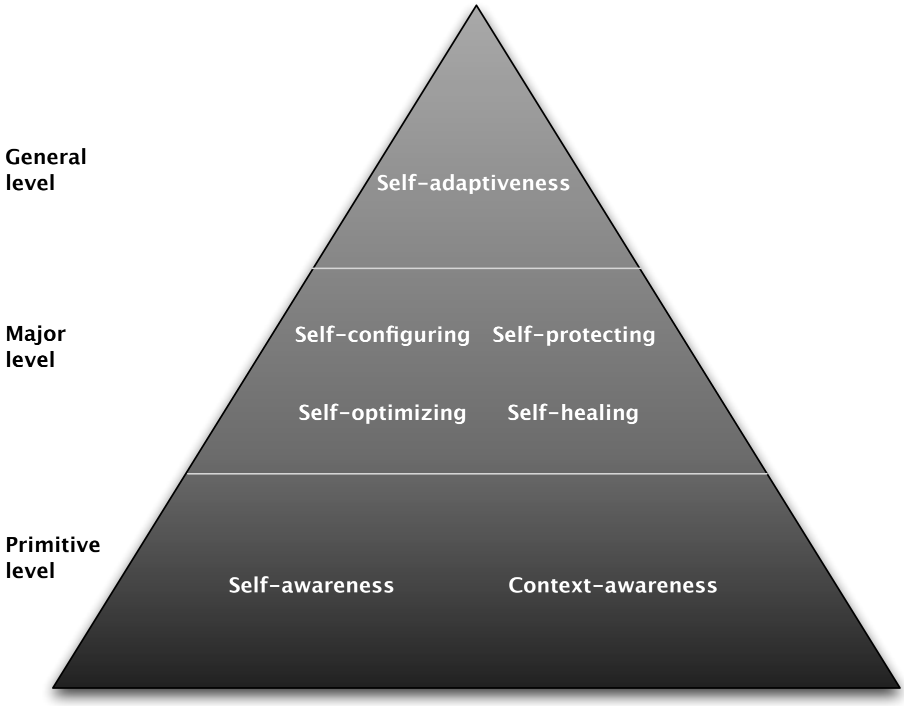


Figure 1.2: Self-* properties hierarchy

Autonomic computing is now researched by several groups around the world and has also witnessed considerable success in the commercial realm [27, 28, 29, 30].

1.3 Involved disciplines

As highlighted by [22], autonomic computing is intrinsically *interdisciplinary*. It does not belong to any particular field but it is indeed cross-cutting research.
Some of the involved areas are:

- *Software Engineering*, which offers several concepts that can be used in self-adaptive computing, such as: indicators to measure the *quality* of the software, requirements engineering (formal methods might be used to define and properties the system must satisfy), Component-Based Software Engineering (CBSE) might be applied to obtain modularity and reusability, Service-Oriented Architecture (SOA) might be employed to help the composition of loosely coupled service

- *Artificial Intelligence* might be applied to make self-adaptive softwares flexible and able to learn from the experience. It might make software able to recognize patterns and boost its ability to decide applying planning, reasoning, and learning techniques. For instance, goal-oriented requirements might be achieved through the use of *agent* and *multi-agent systems*; *Machine Learning* and *Soft Computing* might lend the concept of the "achieving" approach, which learns the best way to act analyzing the stream of sensed data; *Reinforcement Learning* has already been used for dynamic action selection [31] and decentralized collaborative self-adaptive software [32]; moreover *decision theory* and *decision theoretic planning* (such as the Markov Decision Process and

Bayesian Networks) can enhance the decision process.

- *Control Theory* lends the important notion of "feedback loop" that is the base for the sense-plan-act loop described in Section 2.1.1.

- *Network and Distributed Computing* can contribute to self-adaptive computing with the concepts and theory developed in highly-scattered systems. *P2P networks*, *policy-based management* and *Quality-of-Service (QoS) management*, *virtualization*, *hearbeat* and *pulse monitoring* [33] are interesting ideas that might be reused in the context of autonomic computing.

- *Computer architecture* is particularly involved when considering hardware-based solution for adaptivity.

Chapter 2

State of the Art

"Nothing is wonderful when
you get used to it."
Ed Howe

In this chapter we analyze the way autonomic computing systems are in general implemented and what work has been carried out in the last few years in this field. Many researched have developed interesting projects that have some similarities with the work that will be presented in the next chapters.

2.1 Characterizing autonomic computing systems

2.1.1 General concepts

In Figure 1.1.2 we have introduced the concept of *closed-loop* approach, meaning that autonomic software executes and, at the same time, monitors itself and the surrounding environment. This technique is used for multiple reasons: for example, a certain condition might be checked periodically and, if met, a certain action might be triggered; or maybe a certain opti-

mization has been introduced and the systems want to inspect whether it is fulfilling the expectations.
In general we can say that there are certain elements that are *intrinsic* to autonomic computing systems:

Sensors They are the elements that gather the certain information about the system and the environment. They usually transmit it (i.e.: to a buffer, over the network, etc...) and, before doing that, they sometimes also elaborate it.

Monitoring process It gathers information from different sensors. It might preprocess the collected data in order to delineate patterns and discover indicators.

Detecting process It analyzes what has been gathered by the monitor and discovers conditions, problems, etc...It basically determine *when* and *where* a response is required.

Deciding process Given the results provided by the detector it chooses an action to perform (it might stay idle). It basically determines *what* to do and *how* to act.

Acting process Given the context it picks the most appropriate actuator. It is in charge of translating actions into tasks for the actuators.

Actuators They are the components that actually perform a certain change on the system.

These components form the loop shown in Figure 2.1. This description is very general and it fits the case of a *centralized* implementation. A *distributed* implementation has the same components but a different setup that poses an incredibly vast and interesting research topic: reaching a reasonably near optimal resolution in absence of a central entity. It is beyond the scope of

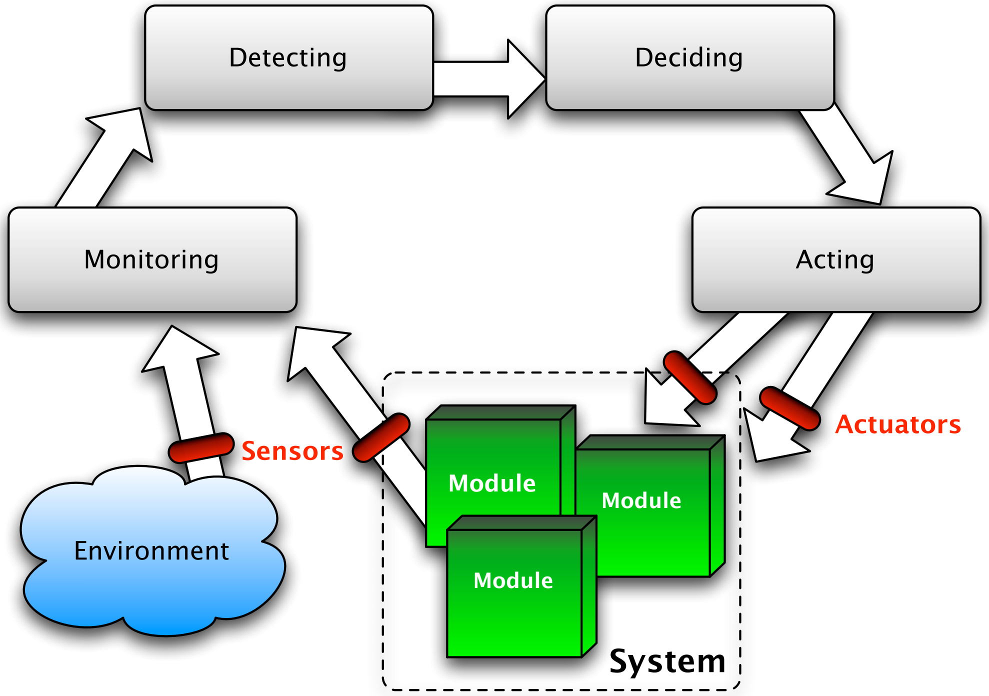


Figure 2.1: Self-adaptation loop

this dissertation to discuss and describe in detail the distributed case, which is left as future work.

The loop in Figure 2.1 is usually called *adaptation loop* [22]. There are several other names that can be used instead: *MAPE-K loop* [34] (shown in Figure 2.2), which stands for Monitoring, Analyzing, Planning and Executing functions, that works thanks to a shared Knowledge-base; or *Observe–Decide–Act (ODA)* loop. The three definitions (*adaptation loop*, *Monitor, Analyze, Plan, and Execute with Knowledge (MAPE-K)*, and *ODA*) will be considered equivalent.

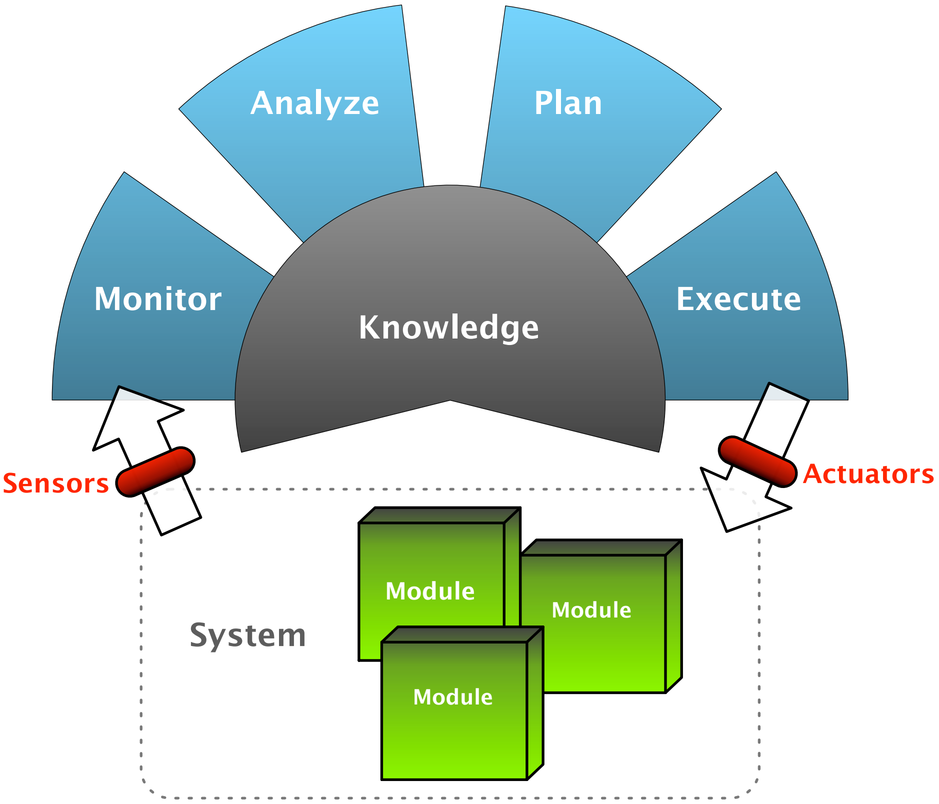


Figure 2.2: A different view: MAPE-K loop

2.1.2 Autonomic objects

This discussion leads to the concept of *autonomic object*, as expressed in [35]. This notion unquestionably helps the modularization of an autonomic computing system. An autonomic object exports at least four interfaces:

1. a *control interface*, which exposes sensors and actuators. It allows external entities to control and monitor the object.

2. an *access interface*, which regulates access to sensors and actuators in order to enforce security. Users are assigned certain privileges depending on the state of the object and on the role of users.

3. a *rule interface*, which makes it possible to add, remove, or edit rules. A rule consists of a condition, an action (defined in terms of *control interface*, e.g.: sensors and actuators) to be performed in case the condition is fulfilled, and an action to be performed in case the condition is not fulfilled.

4. one or more *functional interfaces*

2.1.3 Sensors

Design principles

Sensors always perturb (sometimes only slightly) the system that they help to monitor [36]. For this reason an important principle is to *minimize the measurement overhead*. Some other rules to keep in mind are:

- Sensors should be lightweight

- Sensors should be accessible (potentially remotely accessible)

- Sensors should be reusable

Varieties of sensors

As outlined in many researches [37, 38] there are several types of sensors:

sensors They are small, efficient pieces of code residing within the routine being monitored [39]. There are different kinds of sensors:

> **tracing sensors** They are usually implemented with one piece of code that reside directly in the monitored application. They report every single occurrence of an event within a certain interval of time. This usually happens synchronously with the occurrence of the event. Tracing sensors are used when all occurrences of an event must be known or when actions must be triggered in real-time following the manifestation of a certain event.
>
> **sampling sensors** They are usually implemented with small piece of code that reside directly in the monitored application. They collect information at the request of the monitor. This usually happens asynchronously with the occurrence of the event. Sampling is used when knowing all the occurrences of an event and reacting immediately to a manifestation are not necessary properties.
>
> **extended sensors** These kind of sensors augment tracing or sampling sensors with the capability of preprocessing, analyzing, and computing results. They usually are used in less centralized scenarios where the computation is spread and parallelized in order to reduce the burden on the central monitor.

probes Probes slightly differ from all kinds of sensors that have been presented. They consist of two small, efficient pieces of code: one resides within the monitor and one within the routine being monitored [39]. The fact that probes are implemented as code fragments that reside in the monitoring facility instead of the monitored application provides at least three advantages over sensors:

- They might have a more general view of the system
- The monitored application code is unchanged so that information to be probed can be dynamically defined at run-time
- Some researches show that program perturbation, compared to sensors, is reduced [38]

2.1.4 Actuators

When a certain condition is met certain parts of the system might need to be changed. For a change to be effective there must be a component who knows what to modify and how to do that. Such elements are called *actuators*.

As described in [22] there are a few aspects that can be involved in the change performed by the actuator. The *where* and *what* aspects of the change can be summarized by the following characteristics:

layer an actuator might affect one or more layers of the system.

functionality an actuator might affect one or more specific functionality of the system.

granularity the degree of change might be fine or coarse. It might involve whole components or smaller elements part of a component.

cost and impact Cost describes the impact on execution time, required resources, and complexity of change while impact describes the scope of the repercussions. Depending on how much the cost and how big the impact we can identify two major kind of adaptivity:

- *Weak* adaptation, which performs low-cost or limited-impact actions. Examples of weak adaptation are: modification of parameters, data caching, compressions, load balancing, change of algorithm.
- *Strong* adaptation, which performs high-cost or extensive-impact actions. Examples of strong adaptation are: replacement of whole components, change of architecture, provision of additional resources, redeployment of part of the system.

Monitoring involves obtaining and analyzing performance information from one or more components of the system. Optimization involves using the information to affect either the application or the system. Some examples of actions and actuators include:

- Just In Time (JIT) compiler optimizations [40]
- hot swapping operating system components [41, 42]
- dynamic update of an operating system [5]
- tuning of linear algebra [43, 44]
- redistributing application workloads
- selecting different algorithms based on input data [45] or on more complex factors (such as availability of processors and memory bandwidth) [46, 47]
- loop perforation [2]
- various parameters optimization: available memory, page size [48], OS page replacement policy [49]

- changing cache associativity

Even if we have explicitly divided sensing and acting, both activity usually occur at the same time in order to constantly evaluate the system and the effects of a change.

2.1.5 Characteristics of adaptation

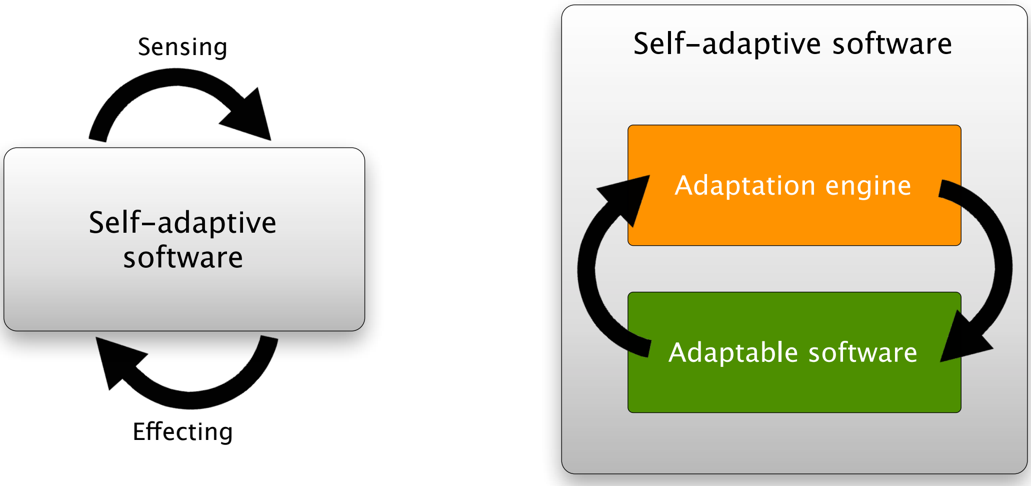
The are several ways to embrace adaptation. The following lists, first proposed by [22], outlines the major differences:

***static* versus *dynamic* decision making** The decisions might be statically defined (e.g.: encoded in a decision tree) or dynamically established at runtime. The parameters that can be involved in a dynamic scenario are: policies [50], rules [51], and QoS definitions [52, 53].

***external* versus *internal* adaptation** Adaptation, in some cases, can be handled entirely by the application itself. This approach is rarely flexible, extensible, evolvable, and scalable. Adaptation often needs global information and for this reason an external approach is often adopted. Moreover an external adaptation engine can be more easily reused. This concept is shown in Figure 2.3.

***making* versus *achieving* adaptation** Self-adaptivity can be introduced in a system at development time, when the system is engineered, or after deployment, through adaptive learning [16]. These situations are referred to with the terms “making” and “achieving”, introduced by [54]. The two approaches are not mutually exclusive.

***close* versus *open* adaptation** Close adaptation has a fixed number of actions to be performed on the system while open adaptation provides the possibility of introducing new behaviors and alternatives.



(a) Internal approach

(b) External approach

Figure 2.3: Comparing internal and external adaptation

***model-based* versus *free* adaptation** When the adaptation process is evaluating a change it might make hypothesis based on a model of the environment and of the system. The model can be extracted in several ways: queueing models [55], architectural models [56], domain-specific models [57], etc.... Sometimes no model is needed and the adaptation process simply knows the goals and the possible alternatives. An example of model-free reinforcement learning has been shown in [32].

***specific* versus *generic* adaptation** The adaptation engine might either address only specific situations or applications or be more generic and not focus on a restricted part of the system.

***reactive* versus *proactive* adaptation** The adaptation engine might either react after an event has happened or try to predict it and behave accordingly. In particular the proactive solution is particularly useful when QoS is involved [53]. However many solutions still do not offer real-time monitoring with QoS guarantees [58, 59, 60].

***continuous* versus *adaptive* monitoring** The monitoring facility might either continuously observe the system or sample a few sensors and enable full monitoring only when irregularities are detected

***human-performed* versus *automated* actuation** Humans might be required to perform some modification on the system. Obviously when the system can actuate most of the changes by itself it becomes more independent and, if trustworthy, more valuable. Manual intervention is extremely common in the *enterprise* domain; many rich monitoring solutions are available (i.e.: HP's OpenView [61] and IBM's Tivoli [62]) but they still require humans to actuate the changes.

2.2 Monitoring and acting in autonomic systems

Autonomic systems as described above are based on the idea of monitoring and analyzing their performance and behaviors. Despite the rapid evolution of programming models and computer systems, performance analysis is still based on these steps [36]:

1. *Application instrumentation*, which enables, either manually or automatically, sensors and probes on the application of interest
2. *Performance data extraction*, which captures data coming from all sensors and probes
3. *Analysis* and visualization of extracted data
4. Reasoning about the extracted data and subsequent *application optimization*

In the last decades we have assisted to a shift from *a posteriori* to *online* optimization. At first, applying (or actuating) a change on a system in order to improve its performance required humans intervention and for this reason it used to be called *steering*. Steering has a long history, especially in the field of super-computers where a complex system had to be monitored and managed along with its long running, resource intensive applications [63, 64]. With the intent of automating such changes, application steering has first evolved into *auto-tuning*: the controlled entity started to self optimize at runtime adjusting statically defined parameters. Auto-tuning has then evolved to support dynamically defined parameters, joining the large family of autonomic application adaptation.

The scientific context has been the most exciting for this advancement since it was the first to adopt parallel systems, which proved to be extremely hard to tune. In this scenario, a good choice of parameters could really boost performance. Moreover, the rising complexity, the always growing users community, the diversity of applications, the number of radically different platforms and architectures targeted for optimization, and the consequent issues in portability and maintainability push toward the automatization of optimizations [65]. For all these reasons parallel systems have long been studied [66, 67] and many projects have been developed: Application Programming Interfaces (APIs) to support both steering and autonomic monitoring [35, 64], a dynamic global resource management system [68], an adaptive application control of distributed applications [36], an instrumentation language for specifying tunable parameters [65], etc. . . .

In this field there is now a trend towards applying all the experience gained in the scientific field to consumer-level products.

2.3 Instrumentation

One of the problem that is linked with the monitoring of a system is *instrumentation*. Instrumentation is the act of inserting either in the monitored application or in the monitor the fragments of code that represent sensors and probes. The term "instrumentation", even if referred to the placement of sensors and probes, will often implicitly implicate the presence of a supporting monitoring architecture.
There are basically two approaches to instrumentation:

- *supported* instrumentation, where the executable is built with the possibility of executing statically or dynamically defined instrumentation code. It will be discussed in Section 2.3.3
- *forced* instrumentation, where the executable is unaware of the instrumentation which is pushed either statically or at runtime. It will be discussed in Section 2.3.1

There are some projects that provide *profiling* without making use of instrumentation. Two well known examples are:

Oprofile [69] It collects system-wide profile information without requiring any instrumentation (or any modification at all) to the compiled executables. It is based on hardware counters and for this reason it suffers the following shortcomings [7, 3, 4]:

1. restricted number of counters
2. sampling delay
3. lack of address profiling
4. even on modern system, impossibility of reacting to single events
5. low level view does not easily translate into high-level application performance indicators

dproc [70] It extends Linux's `/proc` with an efficient, hierarchically-organizer, cluster-wide, application-specific view of monitoring information about both local and remote cluster nodes. Even if its implementation offers high-performance it is not capable of meeting expectations for usages like program steering and adaptation. `dproc` has also been extended to become a highly-configurable QoS-capable monitoring system [53].

2.3.1 Supported instrumentation

This kind of instrumentation is supported by the availability of an API. Programmers place function calls (or code fragments) in the source code of their applications; such snippets will call certain API functions with one of the following purposes:

1. having a certain event logged
2. sending an "I am alive" signal more often called *heartbeat* or *reflex signal*
3. asserting an expectation, for instance a performance expectation

Tracing infrastructures

In the past years the most common research projects provided an architecture to log events. An example of this is *relayfs* [71], a kernel API to transfer data from kernel to user space. Relayfs offers a lock-free implementation that scales well on multi-processor systems.

One of the first most-known tracing facility is the Linux Trace Toolkit (LTT) [72] that helps recording and analyzing complete system behavior. Through

LTT every system event is recorded, making it possible to analyze the functioning of the system itself. LTT provides static instrumentation that causes a non-zero (yet small) probe effect; it defines only a limited number of statically defined events and it cannot take arbitrary actions. Moreover the events have fixed length and timestamp acquisition is inefficient [73].

Even if implemented efficiently ($< 2.5\%$ when observing core kernel events), LTT has been outclassed by the K42 tracing facility. K42 [74] is an operating system developed by IBM and the University of Toronto since 1998. K42 is open-source OS which targets 64-bit multiprocessor systems (especially PowerPC architectures). It uses an object-oriented design to achieve good performance, scalability, customizability, and maintainability. K42 implements each system resource (i.e., an open file or a running process) as an object. Each object reference is stored in a global table, called Object Translation Table (OTT), and this facilitates hot-swapping [42], since every object can be changed on-the-fly. This enables dynamic reconfiguration and adaptability of the system and, when combined with a kernel module loader, supports dynamic update [5] and hot patching of the OS.
K42 features a tracing facility [73, 75] that, just like LTT, has statically defined events. Yet it presents some more advanced characteristics, such as: extreme scalability, variable size events, circular buffer to avoid overflows, per-CPU buffer to perform efficiently on multi-processor systems, lock-less implementation, monotonically increasing timestamp mechanism, events classes, etc.... The K42 monitoring infrastructure is not considered safe because the integrity of traced data is not guaranteed.

DTrace [76] is probably the most advanced tracing technique available at present. It has zero probe effect, it can dynamically instrument both user-level and kernel-level code in a unified and completely safe way. Moreover it is scalable and it supports highly customizable actions (which are described through a C-like language) at any given point of instrumentation. DTrace has been included in three operating systems: Sun Solaris, FreeBSD, and Mac OS X.

Assertions

Assertions augment tracing infrastructures creating a framework that not only collects performance-related data but that also verifies whether performance expectations have been met. Assertion-based projects in fact usually provide an API to be used when programming the target application: through such API users can describe their predictions and their hopes about their application running behavior. For each assertion the runtime system gathers the necessary data and calculates whether the assumption is verified. In an autonomic scenario the system might change its configuration in order to fulfill a certain assertion or it might change its configuration if one or more expectations are not met.
An assertion-based framework is not a new concept: it has been employed in a variety of older [77, 78] and newer projects (such as a self-healing framework [79]).

One of the most prominent assertion-based project is the Performance Assertions (PA) [80] system. PA was born from the need (also described in Section 2.2) of innovating performance analysis techniques: manual reasoning about raw performance data has proved to be tedious, difficult, error-prone, and inadequate considering the scale and the complexity of modern systems. For this reason PA has proposed a new methodology: programmers plant their expectations directly in the source code and the runtime environment gathers the necessary information to verify the assertions. Expressions can contain an assortment of tokens that represent empirically measured performance metrics, constants, variables, mathematical operations, a subset of intrinsic operations, and format specifiers. Responses to failed assertions can take a number of different forms and are customizable through the specification of a user-defined subroutine. PA is dynamic, (quite) portable, and has a limited overhead on runtime execution.

2.3.2 Heartbeats

Many of the solutions presented before provide good flexibility in monitoring a system but they do so at the expenses of simplicity and, sometimes, portability. Starting from this assumption the idea of simplifying sensors, probes and assertions as much as possible was born; this concept evolved into the *Application Heartbeats* (or, more simply, *Heartbeats*) [3, 4], which are a portable, general method of monitoring an application's actual progress towards its goals.

This framework implements a simple yet extremely powerful monitoring infrastructure (the API is made of a small set of functions that makes it straightforward to use). Any application making use of its API has then a standardized method to:

- assert its performance goals registering to the *Heartbeats* and specifying a certain number of parameters: minimum and maximum heart rate, size of the window of observation, size of the heartbeats history buffer, and others.

- update at runtime the progress through the call to a function that signifies an heartbeat. The framework automatically updates all the necessary information about the global heart rate, the windowed heart rate and other internal information.

- monitor the progress of the execution. The calculated information is made available to either external observers or the application itself

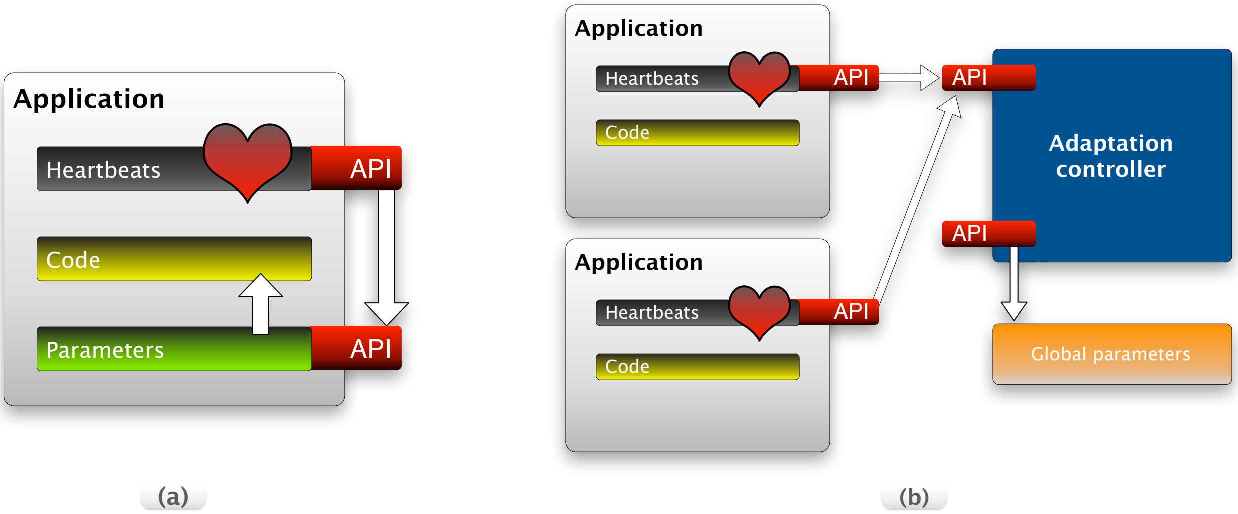


Figure 2.4: Heartbeat API, the two adaptation scenarios: (a) self-optimization of an application; (b) optimization of global parameters by an external observer in a setting with one or more applications.

(as shown in Figure 2.4).

A typical example of use is a video encoder, which might want to be able to deliver 35 frames per second. It then communicates this information to an *Heartbeat API* function and it issues an heartbeat for every produced frame, informing the framework of its progress. An external observer can consequently improve (or reduce) performance through the modification of some parameters, such as the number of cores assigned to the video encoder application.
As shown in Figure 2.4, if needed, the application can impersonate the external observer and monitor itself. Obviously a monitor that resides outside of the application might have access to global information and parameters that are unavailable inside of the application.
The Heartbeat API is thread-aware, meaning that applications can set their performance goals either per-application, per-thread, or both.
The idea of "heartbeat" has its roots in Grid Computing[33], where pulses

where used to constantly monitor the most crucial parts of a system through the emission of a regular "I am alive" signal. The *Application Heartbeat API* builds on this very simple mechanism, leaving untouched its simplicity, lightness, and generality.

2.3.3 Forced instrumentation

Forced instrumentation is a general technique that brings instrumentation to programs without recompiling them and, so, without requiring access (or any modification at all) to the source code. It useful when the source code itself is not available or when there is no willingness or resources to rework it.
For this reason forced instrumentation is the only available option in case we intend to retrofit old applications [81] with self-adaptive characteristics. One example of forced instrumentation can be developed in the context of the *Application Heartbeat API* (already described in Section 2.3.2), where it might be desirable to instrument application that will not be reworked to support the framework. Instrumentation might be automatized with the use of *Intel Pin* [82]. This tool can inject arbitrary code (written in C or C++) in arbitrary locations of an executables. Unlike other solutions, Pin does not statically rewrite the executable with the instrumentation code but it rather adds it dynamically while the software is running.

Pin's approach has been ported in kernel-space by the *KernInst* framework [83, 84], which allows dynamic splicing of (almost) arbitrary code into an (almost) arbitrary location of an executing kernel (Solaris or Linux) without recompilation. This framework work thanks to the `kerninstd` daemon, which dynamically instruments the kernel, and to the `/dev/kerninst` kernel module. In order to write a custom application that interact with `kerninstd`, it is necessary to use the *Kerninst C++ API*. Kerninst has zero probe effect when disabled but it has higher overheads than the K42 tracing facility [73] and it is considered "unsafe" [76] because it might cause fatal errors (in fact it allows users to instrument routines that cannot be instrumented).

Dprobes is a project similar to KernInst and it brings dynamic instrumentation not only to the kernel but, virtually, to any executable. It has zero probe effect when not enabled and it provides a language and a small virtual machine to execute actions in response to certain conditions. Even if DProbes has shown to have some concerns about safety (i.e., invalid loads are handled through an exception mechanism), the way dynamic instrumentation is implemented proves to be lossy when a condition is hit simultaneously on different CPUs. Moreover misemploying DProbes can lead to a system crash. DProbes is mainly used as a (Linux) debugger through which software probes are dynamically inserted into executing programs. When a probe is fired, a user-written probe-handler is executed. DProbes is particularly useful in extreme conditions since it makes it possible to read all the hardware registers, read/write some of them, and read/write any area in the virtual address space that is currently in physical memory.

An analogous project is *DynInst* [85, 84], which aims at manipulating programs at runtime. DynInst is a C++ API that is used to attach to a program and add new bits of code to it. The program being modified does not need to be re-started, re-linked or re-compiled. The API also permits changing subroutine calls or removing them from the application program. The purpose of the DynInst API is to provide a machine independent interface to permit the creation of tools and applications that use runtime code patching.

A relatively recent tool is the Performance Counter Subsystem (PCS), also known as *perf*, that has been introduced in the Linux kernel version 2.6.31.

perf aims at helping both kernel and user-space developers in improving performance of their code, providing a range of mechanisms to determine code portions that need to be changed and highlight bottlenecks in current implementations. The PCS provides an abstraction of special performance counter hardware registers available on most modern CPUs. These registers count the number of certain types of hardware events, such as executed instructions, cache misses, mis-predicted branches, and so on. This happens with a minimal impact on kernel and applications. These registers can also trigger interrupts when a threshold number of events have passed and can thus be used to profile the code that runs on that CPU. In the current release the x86 and PPC architectures are fully supported while support for S390 and FRV is in the works.

Forced instrumentation can also be applied to web applications, just like the case of *AjaxScope* [86], which parses and instruments JavaScript on-the-fly before it is sent to users' browsers. AjaxScope builds on the idea of instant redeployment which is intrinsic to web applications (since every the application is dispatched when the user opens it); in the extreme every single user might have a different version of the application. This is particularly useful for A/B testing, where some users are presented with "version A" of the program and some others with "version B", and for distributed testing, where the burden of logging information for debugging purposes is spread along all the users. This proves extremely useful to gain visibility into errors occurring in users' browsers that would otherwise remain unknown to the developers. This move the debugging capability to the end-user desktop, making it possible to obtain all the information that are available only at the moment of failure [87]. AjaxScope, being implemented as a proxy server, does not require changes on the server-side and does not require installation of plug-ins on the end-user desktop. AjaxScope features *drill-down monitoring* which is a technique that iteratively and *dynamically* *analyzes* the badly performing functions in order to isolate the causes behind a potential malfunctioning. *Static analysis* might be employed as well in similar situations: even if not as powerful as dynamic analysis, it is useful to reduce sensors overhead removing some instrumentation points [88]. A *statistical and dynamical* approach might be adopted to diminish the burden of dynamic analysis.

Other projects (in particular, BrowserShield [89] and CoreScript [90]) rewrite JavaScript "on-the-fly" but with the different purpose of enforcing browser security and safety properties: they might represent an important step in the direction of the self-protecting property.

2.4 Adaptation engine

While almost all of the researches presented before address the problems of instrumenting an application with sensors and developing an architecture that supports runtime dynamic changes, the engine that should make decisions based on sensors data and should command the actuators is not as defined.

Many different approaches have been followed and, in general, it seems hard to design an engine that stay flexible over time. For this reason, AI-related techniques are more and more often employed.

As described in Section 1.3 many different fields might contribute in the creation of an adaptation engine. These are some of possible techniques that might be adopted:

- the *rule-based* approach is very common [80, 35] and it allows for a good degree of flexibility if the operating environment is well known and can be described through rules. Rules guarantee a fairly deterministic behavior and allow actions to be taken in response to the occurrence of an event.

- *decision trees* are generally more static and more hardly extendable than rules but they are usually more efficiently evaluated at runtime.

- *pattern-matching* is an AI concept used [68] to make the program attune to a context that is ever-changing yet is exhibiting some kind of periodic behavior.

- *reinforcement learning* is a sub-area of *machine learning* that might be used to implement the application as an agent aiming at maximizing the long-term reward. Reinforcement learning has already been used in some notable projects [32, 31].

- *Fuzzy logic* is known for working particularly well in the context of conflicting goals and poorly understood optimization spaces [36].

- *Game theory* might also be used to implement a non-centralized system.

Chapter 3

Proposed methodology

"No matter how big and touch a problem
may be, get rid of confusion by taking
one little step toward solution. Do something."
George F. Nordenholt

"When I am working on a problem I never think about beauty.
I only think about how to solve the problem. But when
I have finished, if the solution is not beautiful, I know it is wrong."
Buckminster Fuller

This Chapter provides a general description of autonomic systems and introduces the proposed methodology. We present the design of a modular and flexible *decision engine* capable of interacting with sensors, from which various information about the system are observed, and actuators, which can influence the environment in several different ways.

3.1 Our vision

3.1.1 Overview

Related works show many examples of autonomic implementations: most of them are stand-alone projects, in that they are not part of a complete autonomic system.
Massachusetts Institute of Technology (MIT)'s *Computer Science and Artificial Intelligence Laboratory (CSAIL)*, along with the *Politecnico di Milano*, is working to create such complete system in the next few years. The vision is to modify and extend a Linux distribution with the aim of obtaining an operating system aware of the applications' performance goals and capable of performing changes to itself both in user and kernel space. In this scenario there are many components that are going to be affected:

Kernel-level actuators The kernel might be extended to support mechanisms such as *hot-swap* [41, 42], which allows modules to be swapped at runtime.

User-space libraries New libraries might be added in order to support performance monitoring. The *Application Heartbeats* [3, 4] are a notable example of implemented frameworks, which can be used as the backbone of the monitoring infrastructure.

System libraries In order to improve performance some user-level libraries might be ported to the kernel. Obviously this would make them less portable.

User-level services There exist several software components that have the capability of improving and reducing performance (*i.e.,* CPU frequency scaling). Such *actuators*, which are extremely different and numerous, suggest a standardization of their control interface.

Application They might have to be modified to make use of the new system libraries. There exist approaches (described in Chapter 2) that can be used to avoid to update any single application in the system: instead, monitoring sensors might be injected either statically or at runtime.

3.1.2 A system based on observing, deciding, and acting

In the theoretical view, an autonomic system can be called such when it implements the ODA model, described in Figure 2.1.1. In such model three roles have to be defined and will be described in the following three Sections.
The general realization of this system is shown in Figure 3.1.

Observe

Observation is what makes a system *aware* of itself. Through observation a system understands its state, its current progress, and its possible future actions. This introduces at least three levels of awareness:

1. awareness through the data gathered by sensors
2. awareness of the availability and potential of actuators
3. awareness of the possible targets of actions

Since the aim of the autonomic system is the improvement of applications performance, there needs to be a monitoring infrastructure that gathers the necessary data and fulfills the first kind of awareness described above. To keep track of performance the *Application Heartbeats* (already described in Chapter 2) seem today as the best possible choice: it is a simple yet powerful framework that allows software components (such as applications) to assert performance goals and keep track of the progress.

Other options, such as *Autopilot* [36] (an infrastructure for dynamic tuning of heterogeneous system), *Harmony* [68] (an infrastructure to efficiently execute parallel applications in large-scale, dynamic environments), and *Orio* [91] (extensible annotation-based tuning system which triggers low level performance optimizations depending on the annotations present in the source code) have been considered but either aim at enterprise environments or are not as portable and as flexible as desired.

Regarding the second type of awareness described above, there needs to be a way for the system to find the available actuators and understand their potential in terms of system-wide or application-specific effect. Currently a similar mechanism has not been implemented; yet it will be necessary to introduce it for the autonomic system to be complete.

Similarly, the possible targets aimed at by the actuators must be defined: such information might change dynamically and a mechanisms for determining them at runtime must be developed in order to provide autonomic capabilities.

Decide

The *decision engine* stands at the center of the *ODA* loop. It is the element that exploits the awareness given by the observation mechanisms to elaborate a plan for future behavior. The aim is to improve performance as much as possible making each software component achieve its performance goals.

In particular, the *decision engine*: determines the monitored applications; gathers the information coming from one or more of them; if necessary, updates the list of available actuators; analyzes the performance-related data in order to understand whether to enact a correction policy; possibly decides which actuators to activate or modify; submits the final plan to the actuators.

The role of the *decision engine* has not been implemented at present; yet it will be necessary to introduce it for the autonomic system to be complete.

Act

Once a strategy has been decided, it must be enacted. There are many kinds of actuators that might be available and that might affect performance in different ways: some might affect performance, accuracy, or both; some might impact on the whole system while others might target one or more applications; some might require changes in the application while others operate without the target being aware.

For instance: Unix *niceness* enhances performance leaving accuracy intact while techniques such as *loop perforation* enhance performance altering accuracy, *frequency scaling* affects performance of the whole system while *loop perforation* can target a single application, and so on.

In an autonomic system we can define two main categories of actuators:

Blind Actuators This kind of actuators enact their actions without knowing the actual effect on the system

Conscious Actuators This kind of actuators have an feedback mechanism to be aware of the results of their action

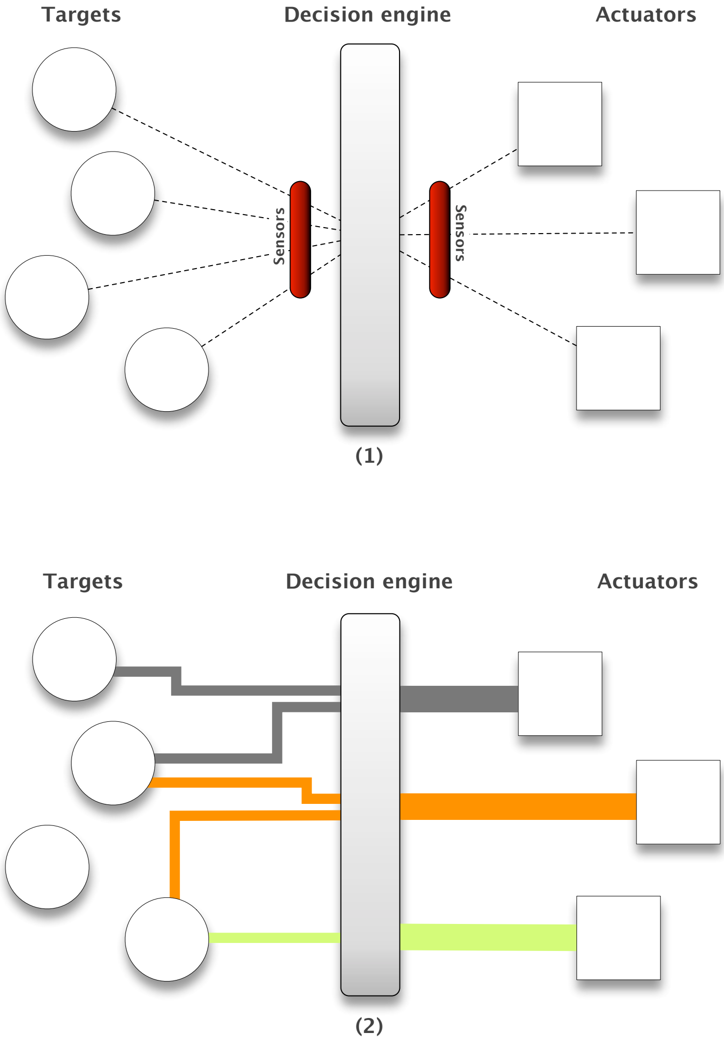


Figure 3.1: *Consensus Object* role: (1) dynamically update the list of available actuators and possible targets; (2) decide which actuator to enable on which target

3.2 Consensus Object: design and surrounding architecture

3.2.1 The need for a Consensus Object

Section 3.1.2 describes the general autonomic architecture and highlights what still has to be implemented.
Those are the components that have to be created to have a complete autonomic architecture:

- A mechanism to discover and dynamically update availability and characteristics of actuators.

- A mechanism to discover and dynamically update the possible targets aimed at by the actuators.

- A central element that: *(1)* analyzes data coming from the applications, *(2)* decides which applications need to be targeted, *(3)* chooses which actuators to activate or to modify, and *(4)* submits the plans to the actuators.

- A *standardized interface* to easily interact with different actuators.

This dissertation aims at making the first steps in the direction of complete autonomic environments. In this context we believe that a central element is needed not just to "conclude" the system but also to orchestrate the various services that would otherwise run independently and, possibly, with very few coordination.
The aim of this Chapter is to design the architecture of a system made of a decision engine, a set of services, the interface to the services, and a mechanism to discover new actuators.

3.2.2 Terminology

This Section presents some crucial definitions. Such terminology is extremely important and will be used throughout this dissertation.

System All the software and hardware components currently in use.

Thread It is an executable element contained inside a process. Multiple threads can exist within the same process; in such case they share resources (*i.e.,* memory), while different processes are independent. The thread is the atomic entity in an operating system that can be given CPU time.

Application An application is a piece of software written to accomplish a specific task.

Process A process is an instance of an application; it might be composed of several *threads* that execute in parallel.

Monitored Process A process that is making one or more entities of the system aware of its performance goals and actual progress. This is possible, for instance, adopting the *Application Heartbeats*. Monitored Process are what has so far been called *targets* in that their performance is altered by *actuators*.

Service It is an extension of what has so far been called *actuator*. Not only does it represent a component capable of performing changes on one or more applications or on the system (affecting their performance) but it also includes the interface to interact with the decision engine and with the application.

Consensus object It is what has so far been called *decision engine*. It has a number of roles:

- It reads the heartbeats coming from one or more applications

- It has a mechanisms to retrieve a list of currently available services; it also is able of updating such list at runtime
- It features a mechanism to choose which services should be used on which terms

Adaptive object Any component in the system that operates through an *ODA* loop.

In the context where applications set their performance goals, the *Consensus Object* orchestrates the available service boosting (or reducing) performance in order to make it possible to achieve the given goals.

3.2.3 Execution scenario

It is possible to envision two different scenarios, where one is the natural evolution of the other:

A. One application (a) and multiple service (s_i with $1 \leqslant i \leqslant n$)

B. Multiple applications (a_j with $1 \leqslant j \leqslant m$) and multiple services (s_i with $1 \leqslant i \leqslant n$)

Other scenarios such as *one application and one service* or *multiple applications and one service* are special cases of *A* and *B*. Within this dissertation we will start our analysis from *A* and we will mainly focus on such scenario. We will try to extend the findings to *B*. The adaptive framework is designed to support multiple applications and multiple services from the very beginning. Nevertheless opposite goals or scarcity of resources might make it impossible to reach the given aims: in these scenarios the proposed methodology will try to maximize the number of applications that are inside the desired heart rate; unfortunately there will be no guarantees that the correct balance of resources among applications will be reached.

3.2.4 A non-centralized approach

The *Consensus Object* represents a centralized approach: it is the central element that gathers all the performance information coming from the applications and coordinates services. Although it represents a *single point of failure*, it is the only approach that allows to have a global overview of the system performance. Moreover, through "smart" decision engine algorithms, it can really boost performance learning the effects of each service from experience. As a last consideration we would like to say that the *Consensus Object* will be designed in a way that, in case of crash, the whole system can keep on running almost normally. For this reason the services will be very independent in many aspects and this makes it possible for them to run also after the *Consensus Object* has abnormally quit. This characteristic is known as *mitigation of the single point of failure*.
An alternative to the centralized approach is a *game-theory-like* scenario where each service acts independently but can collaborate and communicate with other services in order to try to reach an optimal solution. This scenarios is certainly much more complex and currently seems overly elaborate to be implemented effectively. For this reason the centralized approach has been embraced and it has been chosen to develop a first implementation of a complete autonomic system. The *game-theory-like* scenario has not been discarded but it might be considered again once the implementation of the centralized version is complete: we believe that this is the necessary approach to understand which of the two actually works better.

3.2.5 A consideration about performance goals retrieval

In such scenario one uncertainty might arise: who establishes performance goals? And who is in charge of updating the progress towards such goals?

The proposed solution makes use of the *Application Heartbeats*: this means that each application that adopts this framework autonomously sets the goals and updates the progress. This solution is perfectly reasonable since only the application knows what its goals are and when it has finished the computation of a fraction of information.

One might raise the potential problem of applications that deceive the system with intentionally wrong information. This might happen if the application wants as much resources as possible.

Even though this is not an impossible situation, in our opinion it is not an issue because of the following:

- issuing less heartbeats, or providing an impossibly high performance goal defeats the purpose of self-adaptive computing
- the role of the *Consensus Object* is to mediate resources among processes; for this reason enough resources can usually be guaranteed even when one of the processes is particularly demanding
- even in current non-autonomic systems a similar argument might be raised; in fact, a process can create hundreds or thousands of threads, can fork hundreds or thousands of times, or it can "`nice`" itself to obtain as much resources as possible

In response to these potential issue we have developed *Pacemaker*, which is tool based on Intel's *Pin*. It analyzes the execution patterns of a given process, finding the loops where most of the execution time is spent; it then injects the code to produce heartbeats, making the given application (even if closed-source) heartbeats-capable. This way it is possible to circumvent the issue of an application issuing fraudulent heartbeats.

3.2.6 Services

There are several different elements that can affect performance on a system. Such elements usually come in different forms and bring different consequences. This Sections presents a list of possible services and proposes a first classification.

Application "knobs" They are parameters in an application that can be changed, causing an alteration in performance. An example is the *number of threads*: there is not a best value overall but, depending on the load of the system and on the availability of computational cores, a wisely chosen value can improve performance. This usually is an application-specific optimization.

Application implementations Many applications make use of certain algorithms to carry out their duties; depending on the scenario a different implementation might lead to better results. For this reason a way to enhance performance could consist in changing the implementations of a certain algorithm. A new implementation of the swapping algorithms certainly needs to take into account the experience with K42 [42, 74, 92, 93]. This usually is an application-specific optimization.

Core allocation Under modern operating systems it is possible to force a given process to be run only on certain processors or cores. Ideally, when each process is coupled to a certain core and when the number of cores is sufficiently high, there would be no need for time sharing among processes. This optimization is enabled on a per-application basis.

Memory allocation The *core allocation* service is particularly useful when used on processes that are *CPU-bound* or *CPU-intensive*. There exist another category of processes that requires, instead of processing resource, memory. Such programs are said *memory-bound* or *memory-intensive*: in order to be optimized they require memory to be allocated easily and efficiently. This optimization is enabled on a per-application basis.

Niceness adjusting Modifying the niceness (which is the priority of a process) can enhance or reduce performance of a given application. This optimization is enabled on a per-application basis.

Locks Changing the policies on the management of locks might also affect performance , as shown in [1].

Frequency scaling It is a widely popular service that reduces processor clock frequency slowing down the computation on the entire system. When using such optimization it is of course necessary to take into account its global effect.

Dynamic perforation Some optimizations increase performance to the detriment of accuracy of calculation. For instance, this is the case with dynamic loop perforation [2]. This optimization is enabled on a per-application basis.

Those service can be classified in the simplest possible way: using the notion of *application-awareness*. Two categories emerge:

- services that can be enabled without the application being aware of them. Most of the services listed above are categorized this way. Services belonging to this category are: *application implementations, core allocation, niceness adjusting, locks,frequency scaling, dynamic perforation.*

- services that require the program to be aware of them. At present the only service that has been identified to be part of this category is represented by the *application "knobs"*.

One might not understand because *application implementations* are part of the first category. These two definitions will become straightforward once the communication infrastructure is analyzed in Section 3.2.7.

3.2.7 Communication infrastructure

In order to understand the communication infrastructure it is necessary to highlight the *information flows* (which will be called FN from now on) that have to take place in this autonomic system

F1. the *Consensus Object* has to know which applications make available their performance goals and their progress towards them

F2. the *Consensus Object* has to know about the heart rate of the application

F3. the *Consensus Object* has to know which services are available

F4. the *Consensus Object* has to control the services

F5. the application has to advertise the parameters (*application knobs*) that it intends to expose

F6. each service has its own way to actuate a change. Some involve the communication of some kind of information back to the application; in the *application knobs* example the values of one or more parameters has to be communicated.

F7. possible implementations of a functionality must be known to some component in the system

The described information has to be delivered in order for the autonomic system to be effective. Flows *F1* and *F2* will be described in Section 3.2.7. Flows F3 and F4 are delivered through apposite interfaces which will be discussed in Section 3.2.7. Flows F5, F6, and F7 can be delivered

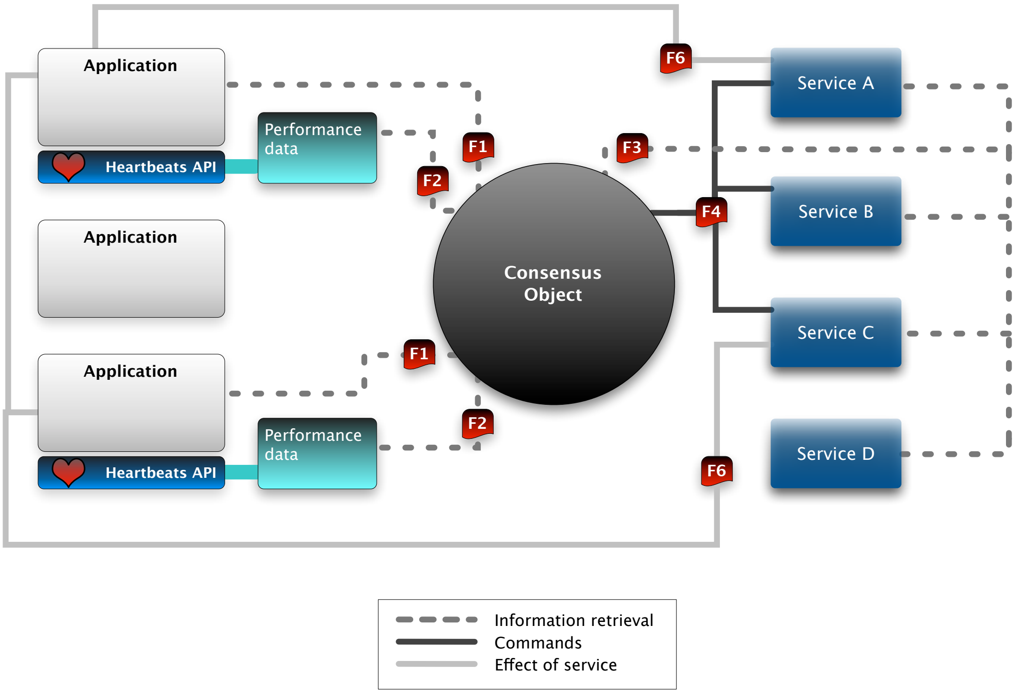


Figure 3.2: General concept behind informations flows F1, F2, F3, F4, and F6 in a multi-service and multi-application scenario

introducing two libraries (*Implementations library* and *Parameters API*) that will be discussed in Section 3.2.8.

Information flows F1, F2, F3, F4, and F6 are shown in Figure 3.2 (F5 and F7 are not shown because they require concepts that will be introduced only later).

Monitored applications information flows

For an application to be able to communicate its goals two steps have to be performed:

1. the *Consensus Object* has to become aware of the fact that one or more processes are making use of the *Application Heartbeats* to set their goal and update their progress; this is possible thanks to F1

2. the *Consensus Object* has to be able to access the information recorded through the *Application Heartbeats*; this is possible thanks to F2

F1 is also known as *application registration* and it might be implemented exploiting the fact that, for each application that makes use of the framework, the *Applications Heartbeats* create a temporary file that is deleted when the process either terminates or unregisters from the *Applications Heartbeats*.
F2 is straightforward since the *Applications Heartbeats* share a memory segment where data is stored and makes it possible to access such information with an apposite set of functions: through them it is possible to set an "heart rate monitor" that enables the *Consensus Object* to read the collected data and statistics.

Service registration and control interface

Any given service can become active only after two steps are performed:

1. the service must advertise its existence to the *Consensus Object*
2. the service must implement a standard interface in order to receive commands from the *Consensus Object*

In the two following Sections these two steps will be thoroughly explained.

Service registration The *Consensus Object* has to be aware of the services that are available in the system and it also has to know a few information to characterize them. It should in fact know the main traits of the service, such as: name, granularity (whether it affects the whole system or can target single applications), and so on. In order to exchange this preliminary information a common location must be defined. Given that:

- the *Consensus Object* and the services run as separate process

- their startup order is not known (we do not assume the *Consensus Object* to be already running when one or more services start)
- the PID of the *Consensus Object* and the services are not known *a priori*

it is necessary to hard-code in both the *Consensus Object* and the service a common location. This can be accomplished exploiting *files* or *named pipes*: for instance, in a folder one file per each available service might be written (similarly to what also happens in the *Applications Heartbeats* for the monitored applications); or a named pipe might be used to write and read available services.

Through this preliminary communication channel the service might inform the *Consensus Object* of its name, PID, and all the information needed to establish a new channel where to exchange control directions. The chosen solution should be flexible in that it should work independently from the startup order of the *Consensus Object* and the services: *files* and *pipes* seem to be a good answer since the service advertises its existence right after startup. The *Consensus Object* periodically reads from it and updates the list of available services. If the *Consensus Object* has not started already it will read the list right after its startup.

The registration mechanism is identical among each service: such characteristic is necessary to keep the implementation as modular, expandable, and flexible as possible.

Service interface Registration is necessary to establish the communication channel through which the *Consensus Object* controls the service. Such channel might be arranged as a *shared memory segment*. In this scenario the initial communication should be used to exchange the key to be used to access the shared memory segment, which has already been allocated by the service. When the *Consensus Object* needs to control the service it

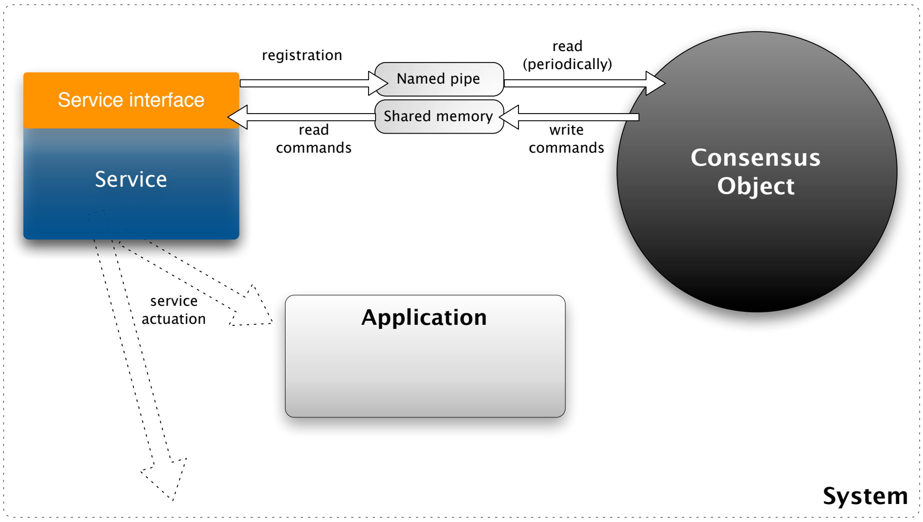


Figure 3.3: Service registration and service interface

simply writes to such segment, specifying a high-level command and a target. Examples of commands might be `increase_performance` and `decrease_performance` or `enable_service` and `disable_service`. An example of target might be the Process Identifier (PID) of a process, which has to be affected by the service.

The PID has to be communicated because the service, in most cases, will be very general and might target different processes in different ways. For instance, dynamic perforation might be used on a process and not used on others. Obviously, this kind of information is not needed in case the service has an impact on the whole system.

Each service, regardless of the kind of optimization it performs, is treated in the same way by the *Consensus Object*, which commands the service through high-level commands. This way the definition of service remains particularly general and allows a great degree of *modularity*: future services can be implemented without necessarily updating the *Consensus Object*.

Figure 3.3 shows what was explained in this Section.

3.2.8 Autonomic libraries

Information flows F5, F6, and F7 can be explained after a short premise. One of the purposes of autonomic computing is to reduce the burden on programmers, hiding some of the complexity they would have to deal with if they had to implement a self-adaptive system from scratch. Some possible uses of autonomic computing seem particularly promising: for instance, the capability of self-tuning some crucial parameter of the system (*i.e.,* the number of threads in an application); or the capability of choosing at runtime the best implementation of a given functionality.

These two examples should be made available to the programmer with the least effort on his or her side. Our vision is to provide such functionalities introducing two libraries: the *Implementations library* and the *Parameters API*. Following this approach, newly built applications can make use of them and easily become part of the autonomic system.

Libraries are services themselves. In order to make the optimal choice they have to communicate with a central entity, the *Consensus Object*, that has a global view on the system. For this reason both the libraries register to the *Consensus Object* and implement the service interface. The *Consensus Object* commands will then be translated in meaningful parameters or in a possible implementation switch.

Implementations Library is a general term to refer to a library that exposes some functionalities and has the capability of switching among implementations. Suppose an application needs to sort a very big set: instead of writing the code to perform such action the programmer might rely on an external library (that obviously exposes the required methods). Since there are many algorithms that order a set and they all perform differently, it would be desirable to have the library automatically pick the best one. Obviously abstracting such complexity from the programmer is highly advantageous. Yet, for a correct choice of the implementation the library must

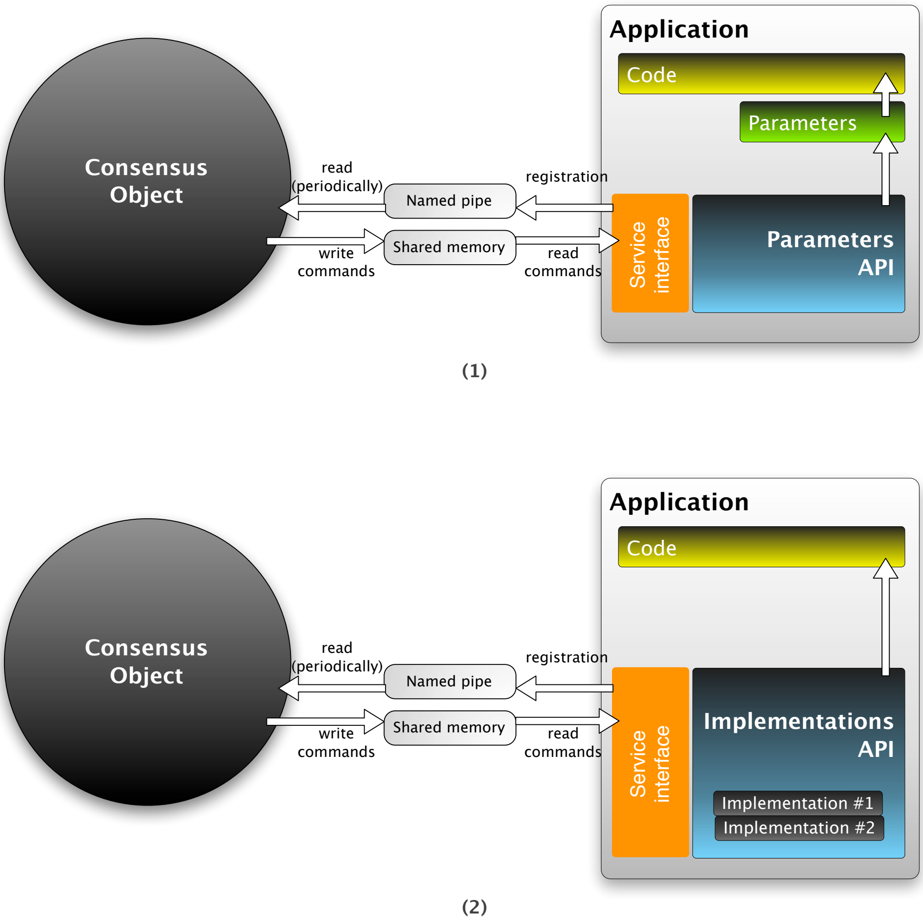


Figure 3.4: (1) Implementations library; (2) Parameters library

have data regarding performance. For this to be possible it has to open a channel with the *Consensus Object*, register as a “normal” service, receive the commands, and obtain the necessary information about performance exploiting the *heart rate monitor*. The *Consensus Object* will then control the library just like it would do with any other service, through high-level commands. The library knows the available implementations (F7 is then totally internal to the library) and, given such information, picks the best one.
The mechanism that handles the change of implementation is called *hot-swap*. *Hot-swap* can be defined as the dynamic insertion and removal of code in running systems. It usually consists of two pieces:

- *Interpositioning*, which involves inserting additional components between existing ones; such components usually assist the transition.
- *Replacement*, which allows an active component to be switched with another component while the system is running, and while applications are using the resource managed by the component. This allows components suited to a particular environment to be switched as conditions change and allows new upgraded components to replace existing ones.

Hot-swap is a technique that has been widely discussed in recent research studies, especially those concerning the K42 Operating System [42, 74, 92, 93, 73]. Its aims can be numerous: security patches, bug fixes, introduction of additional system monitoring, testing, and so on. In our work we will make use of *hot-swap* only to deal with implementation changes.
The whole K42 makes an extensive use of object-oriented technologies and each resource is managed as an object instance. Each object in K42 is mapped in a particular table known as OTT. Such table is the basis for *hot-swap* since simply replacing the reference to an object makes it possible to change the implementation of the given object. Yet there are many aspects that have to

be taken into account to ensure that the transition is smooth and guarantees data integrity. For this to be possible K42 researches have identified three essential issues:

Quiescent state Before it is possible to *hot-swap* the involved components must be brought into a safe state. The swap can only be done when the component is not currently being used. One way to achieve a quiescent state is to require all clients of the component to acquire a reader-writer lock in read mode before any call to the component. However, this would certainly add overhead for the common case and the lock could not be part of the component itself.

State transfer When it is safe to perform the *hot-swap*, the system has to decide what state needs to be transferred and how to transfer it to the new component. Such transferral should not only be safe but also as efficient as possible.

References swap In case the swapped element is referred by its clients (for instance, to allow bidirectional communication), the system has to know how to swap all of the references held by the clients of the component so that the references refer to the new component. In a system built around a single, fully typed language (for instance, Java), this could be done using the same infrastructure used for the implementation of the garbage collector. However, this might prove to be prohibitively expensive when dealing with few components switches. An alternative would be to partition a hot-swappable component into a *front-end* component and a *back-end* component, where the front-end component is referenced (and invoked) by all the clients, and its only function is to to forward requests to the back-end component. There would then be only a single reference (inside the front-end component) to the back-end component that would require to be changed when a component is swapped. This kind of implementation seems reasonable but it of course adds extra overhead to the common case.

K42's proposed methodology is then based on the following steps:

1. instantiate the replacement component
2. establish a quiescent state for the component to be replaced
3. transfer state from the old component to the new component
4. swap the new component replacing all references
5. deallocate the old component

A simple representation of this procedure is shown in Figure 3.5.

An example of *Implementations library* has been described, implemented, and tested as part of this work: as it will be described in Section 4.2, the context of use provides the opportunity of simplifying the swap mechanism making use of a shared data structure and thus moving away from the more general case.

Similarly the *Parameters Library* is an API to expose certain parameters to have them controlled by an external entity. Since it is not advisable that the *Consensus Object* knows every possible parameter (and the effect that each of them can have on the system), the library will simply register to the *Consensus Object* as a service and implement the *service interface* to be controlled by the *Consensus Object*. It will then translate high-level *Consensus Object* commands into changes in the parameters. F5 is then a flow going from the application to the library and F6 is going in the opposite direction. Figure 3.3 shows the interaction between the *Consensus Object* and the two proposed libraries. Figure 3.6 summarizes what has been discussed so far: for each information flow identified above a possible implementation is given.

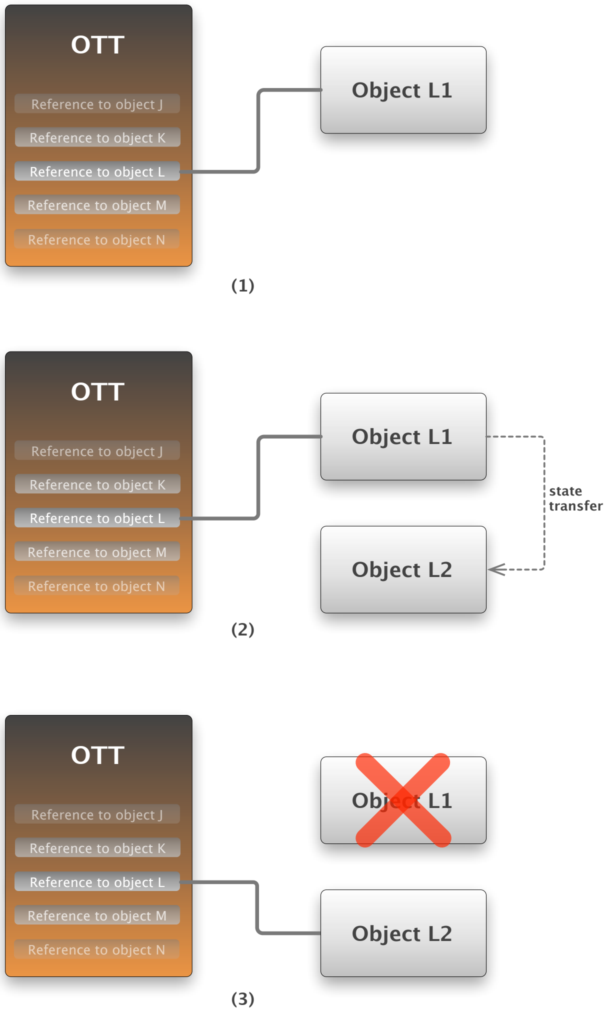


Figure 3.5: (1) Normal operation with object L1 (that is referenced in the OTT). (2) L1 has to be replaced by an improved or optimized version called L2. First of all L2 is created, L1 reaches a safe state, and then L1's state is transferred to L2. (3) References are updated and L1 is deleted.

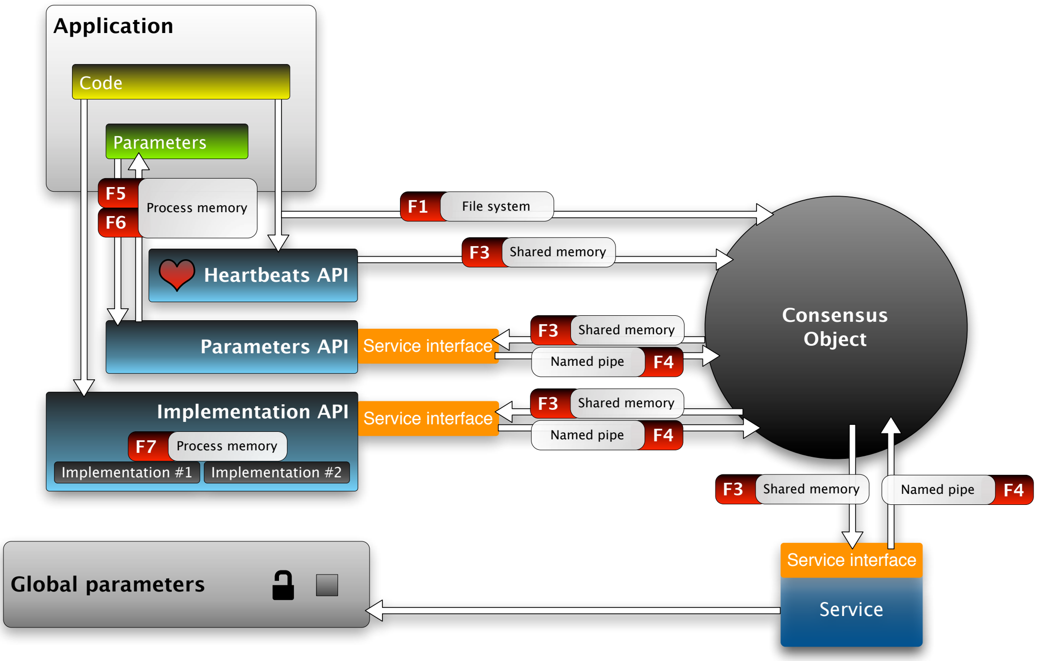


Figure 3.6: Global view of the information flows. The picture shows potential ways to deliver the flows defined in Section 3.2.7

3.3 Decision engine

The *Consensus Object* has the responsibility of deciding which services to activate on a given target. It should also be able to decide when to disable a service. The *Consensus Object* does not know which is the potential of each service (*i.e.,* , how much a given service can improve performance) and, more in general, it does not have a thorough description of the service characteristics. The *Consensus Object* at most knows whether the service can improve performance (in such case it is an *UP* service), decrease performance (in such case it is an *DOWN* service), or both (in such case it is an *UPDOWN* service). This means that, at least at the beginning of the execution, all services are equal to the *Consensus Object*. The ideal algorithm would have to evince from past behavior what services are most appropriate in a certain situation. This would of course make it easier and

transparent to the services to identify services clashes, caused, for instance, by a contention on resources.
Such "smart" algorithm would have deep roots in *Machine Learning* techniques and would certainly requires quite a big effort to develop. Since the focus of this dissertation is on developing the infrastructure for a complete autonomic system, we have first implemented a simple decision engine based on a heuristic; we have then designed and implemented an substitutive algorithm that is more complex, flexible, and capable of learning from experience.

3.3.1 Decision tree

The first algorithm that has been designed and implemented is based on a simple set of rules: we believe that it works well in the base case and help us to deliver not a finished product but an *enabling technology* that can be used as the foundation for future projects. The functioning of the algorithm is based on a static *decision tree* (shown below):

- The *Consensus Object* goes through all the available processes; please note that, as explained in previous Sections, this implementation of the *Consensus Object* is multi-application aware but it does not make any particular consideration when more than one application is monitored. This means that in the multi-application scenario an optimal solution is not guaranteed.

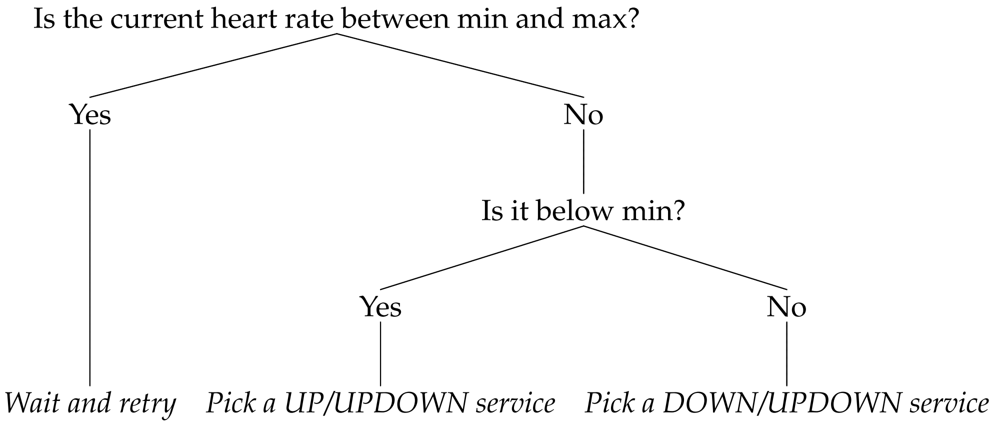
- When an application is under-performing (or over-performing), all the available services are scanned to see whether there exists at least one that can be enabled (or disabled) to make performance return in the application's range of heart rate. If it exists, the *Consensus Object* sends an appropriate command to it.

There exist several types of commands that the *Consensus Object* can issue:

ON Turns the service on. The service usually starts its action and also starts monitoring the heart rate in order to *try* to make its action conciliate with the obtained results

OFF Turns the service off.

DOWN_BY (and UP_BY) They are commands that might be useful in the multi-application scenario. The purpose of *DOWN_BY*, for instance, is to ask the service to reduce its action. This might be useful to either make the service release resources or make it intentionally underperform to reach a global optimum. In fact, when multiple applications are running, some applications might not be able to reach their goals: in this case one application might have to be penalized in order to make the others closer to their minimum heart rate. The way this commands work is by forcing the service to believe that the heart rate goals has been reduced by a certain percentage.


Is the current heart rate between min and max?
Yes
No
Is it below min?
Yes
No
Wait and retry
Pick a UP/UPDOWN service
Pick a DOWN/UPDOWN service

3.3.2 Reinforcement learning

Machine learning is an extremely wide field and it has given birth to many techniques that contribute to making current computing systems capable of interacting and learning from the surrounding environment. Such techniques include, for instance, *supervised learning* algorithms, which work by trying to mimic the training set; the given training input is indeed used to infer a general policy for generating outputs. An example of a supervised learning algorithm might be the prediction of house prices in a certain city; in this case the training set would be a made of examples of house prices (associated to certain characteristics needed to infer the model).

On the other hand *unsupervised learning* algorithms are able to gain knowledge about the environment without being explicitly guided by a training set; the algorithms in this category infer a general policy from observation, not from examples. An instance of a *unsupervised learning* algorithm could be the clustering of data in order to classify different kinds of elements.

A different approach is given by *reinforcement learning*, which discovers the best actions to undertake through evaluation of the performance. Reinforcement learning differs from supervised learning because no right or wrong answers are given: the learning process is guided only through a reward function that indicates when the agent is doing well or when it is performing poorly. For instance, if we were considering the control system for an inverse pendulum, initially we might not know the “correct” actions not to make it fall; in this case the reward function might give the robot positive rewards for standing still or moving forward and backward, and negative rewards for falling over. The algorithm will then figure out how to choose actions over time so as to obtain large rewards. Reinforcement learning has been successfully applied to many problems in many different fields and we believe it might suit the self-aware self-adaptive context.

We start by defining the Markov Decision Process (MDP), which is a fundamental formalism to understand reinforcement learning. A MDP is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S is the set of states. It is important that the state includes both information about current sensors measurements and past "memories". While it is of course not required that the state represents *everything* about the environment, information about past observations might be retained to improve the quality of decisions. In short, "we don't fault an agent for not knowing something that matters, but only for having known something and then forgotten it" [94].
- A is the set of actions
- $\{P_{sa}\}$ define the probability of shifting from a state s to a state s' when a certain action a is taken. Basically $\{P_{sa}\}$ gives the distribution over what states we will transition to if we take action a in state s
- $\gamma \in [0, 1)$ is the discount factor

The dynamics of a MDP proceed this way: an initial state $s_0 \in S$ is picked and some action $a_0 \in A$ is chosen; after taking the action the state will transition (following the distribution probability defined by $\{P_{sa}\}$) to a new state s_1. From here a new action a_1 is chosen, and so on. This can be described by the sequence:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} ... \tag{3.1}$$

Whenever a new state is reached, a reward might be given to the learning agent. The reward accumulated over all the states transition can be expressed as:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ... \tag{3.2}$$

Since we aim at optimizing the obtained reward we might say that we want to maximize this quantity:

$$E[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + ...] \tag{3.3}$$

Some important definitions are:

- A *policy* is a function $\pi : S \mapsto A$ that maps states into actions

- The *state-value function for policy* π, known as $V_\pi(s)$, is the expected sum of rewards (discounted by the factor γ) received from starting from state *s* and then following the policy π

- The *action-value function for policy* π, known as $Q_\pi(s, a)$, is the expected value of the sum of rewards (discounted by γ) starting from a state *s*, taking action *a*, and then following policy π

- given a policy π, its state-value function satisfies the *Bellman equation*, which states that V^π is made of two components: the immediate reward and the expected sum of future discounted rewards

$$V^\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi}(s')V^\pi(s') \tag{3.4}$$

- the *optimal state-value function* is defined as:

$$V^*(s) = \max_\pi V^\pi(s) \tag{3.5}$$

  which is the best possible expected sum of discounted rewards

- the *optimal policy* is defined as:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s')V^*(s') \tag{3.6}$$

Given the problem described in the previous sections we can make the following considerations:

- the *states* S are given by the current performance states of applications (*i.e.,* above performance, below performance, in range, and so on) and by the current states of services (*i.e.,* service S_1 is globally enabled, service S_2 is enabled only on a certain application, and so on)

- the *actions* A are all the possible activation/deactivation of services either globally or on a specific application. Please note that the set of available actions depends on the current state since, for instance, if a service is already enabled on a certain target, it might *only* be disabled. Moreover it might be necessary to consider a *null action* which represents the decision of not taking any action

- the transition probabilities $\{P_{sa}\}$ are not known and they have to be estimated at runtime

- a balance between *exploration* (choosing actions almost randomly in order to discover potential benefits) and *exploitation* (choosing actions that in the past have brought high rewards) has to be reached. Two methods are typically used:

  ϵ-greedy With probability ϵ this algorithm chooses to explore instead of exploit; exploration actions are chosen randomly and are all given the same probability

  Softmax Each action has a probability of being chosen that is proportional to the value of the action-value

- the reward might be given by a function of the performance of applications

In this context the *R-learning* [95] algorithm seems to fit particularly well our problem. R-learning has the following characteristics:

- it is part of the *Temporal Difference* algorithms family, which works without needing a model of the environment

- it is an off-policy algorithm, which means that the policy used to generate behavior (known as *behavior policy*) might be unrelated to the policy that is evaluated and improved (known as *estimation policy*)

- it does not divide experience into episodes, allowing for continuous learning (even when terminal states are reached). This also means that there is no need to discount
- the behavior policy can either be ϵ-greedy of Softmax; in our implementation we have chosen Softmax

The pseudo code of the algorithm is given below:

```
Initialize Q(s,a)
Initialize rho
Repeat forever:
  s <- current sate
  choose action a in s using behavior policy (softmax)
  take action a, observe r and s'
  Q(s,a) <- Q(s,a) + alpha[r - rho + max +
              max Q(s',a') - Q(s,a)]
  If Q(s,a) = max Q(s,a) then:
    rho <- rho + b[r - rho + max Q(s',a')
            - max Q(s,a)]
```

Listing 3.1: Pseudo code for the R-learning algorithm

3.3.3 Decision policies

An important topic that has not been raised in previous Sections is the need for good *decision policies* that should drive both the *Consensus Object* and the services. In fact there are many decisions that can be taken either with an heuristic or with a more elaborated calculation.

The *core allocator* example is certainly emblematic. When performance drops under a certain threshold (*i.e.,* the minimum heart rate specified by the application) it is needed to decide how to react to improve performance. To increase performance more cores could be allocated to the application; unfortunately the new number of cores assigned to the application is not

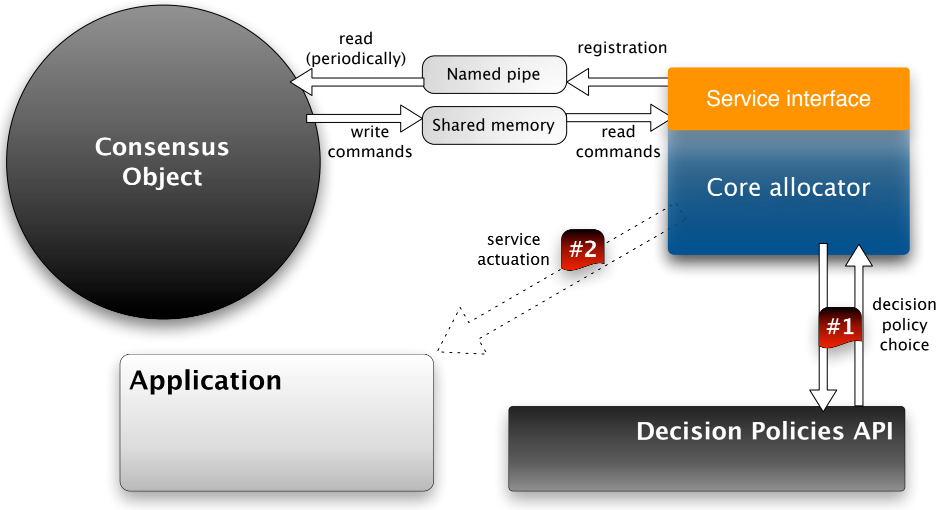


Figure 3.7: The API that gathers decision policies

straightforward. It might be enough to allocate one core in addition to the ones already allocated; yet maybe the decision policy might exploit what is known from the *control theory* and use a PID controller.

It seems reasonable to gather all these *decision policies* in an additional API that the *Consensus Object* and services can access. Figure 3.7 illustrates the idea.

Chapter 4

Proposed Implementation

"When the solution is simple,
God is answering."
Albert Einstein

In this Chapter the proposed implementation will be thoroughly described. First of all, an example of *Implementations Library* will be outlined. Then, more details on the *Consensus Object* and *Services API* will be provided.

4.1 Implementation considerations

Throughout this Chapter some principles will be constant in every implemented program. Whenever possible the realizations will be as modular, extensible, flexible, portable, and maintainable as possible.

4.2 Data Encryption Standard (DES): an example of Implementations Library

This Section presents an FPGA-based implementation of an autonomic system which features several *self-** properties: *self-awareness*, *self-reconfigurability*, and *self-optimization*.

4.2.1 General description

This implementation of an FPGA-based self-aware adaptive computing system blends techniques developed in different research fields, such as monitoring, decision making, and self adapting. The chosen approach merges the potential brought by reconfigurable hardware with the state-of-the-art in performance assertions, monitoring, and adaptation: FPGAs offer tremendous computational power while the possibility of running an operating system on top of it makes it possible to observe performance and take actions that can involve both software and hardware adaptation.
We have developed a mixed hardware/software system that monitors performance through apposite software components while providing both hardware and software implementations of the same functionality.
The system is in charge of choosing at runtime among the set of possible implementations (hardware or software among the available implementations) according to different criteria, such as expected performance, available area (set of resources) on the FPGA, input data type and size, functionalities already implemented and available as hardware components.
The purpose of the proposed architecture is to encrypt data using the *DES* algorithm as fast as possible. For this to be possible the system must be aware of the possible paths of executions and must choose not only with the static information that it might have but also with the total set of dy-

namic (calculated at runtime) data and statistics. A complete *ODA*-based (refer to Section 2.1.1 for more information) system was designed:

- *observation* is carried out through the *Application Heartbeats*
- *decisions* are taken through a simple *adaptation engine*, which is fed with the data gather through the *Heartbeats*
- *actions* are obtained through the *hot-swap* of the available implementations

In particular the system features two different implementations of the DES algorithm: a software and a hardware implementation.
The three *ODA* phases have been implemented within the proposed system, by using different software libraries. Figure 4.1 presents the overall structure of the proposed system.

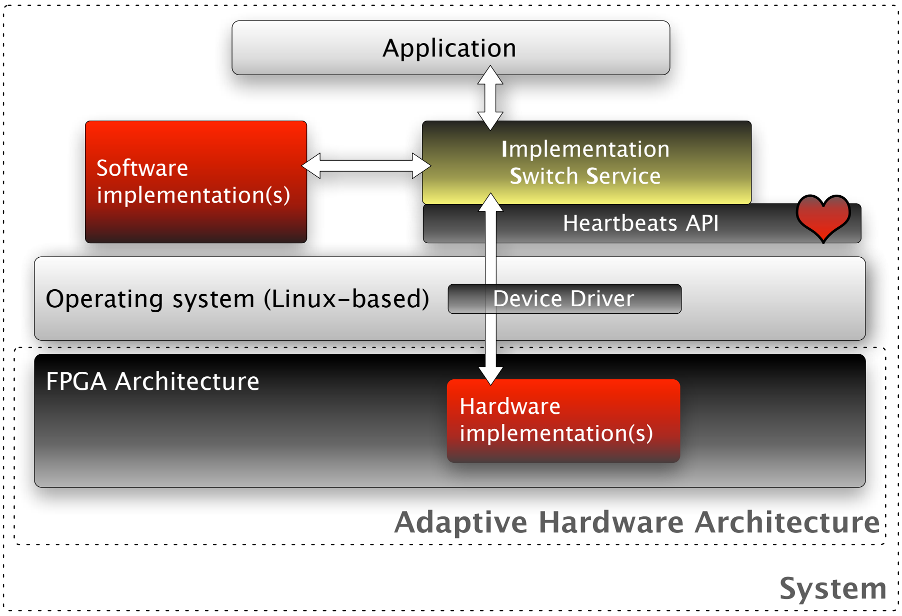

Application
Software implementation(s)
Implementation Switch Service
Heartbeats API
Operating system (Linux-based)
Device Driver
FPGA Architecture
Hardware implementation(s)
Adaptive Hardware Architecture
System

Figure 4.1: Self-Aware Adaptive computing system

4.2.2 Implementation Switch Service

In the presented implementation the component capable of analyzing the data gathered through the *Heartbeats*, taking decisions, and performing actions is called Implementation Switch Service (ISS). It queries the *Application Heartbeats* framework to have knowledge of the progress of the execution and obtain an overall history of the heart rate. Fed with such information, the decision mechanism chooses at runtime the best implementation to use in accordance to given constraints (such as the need to avoid oscillations) and goals (such as the desired heart rate).

The need for a dynamic choice between available implementations is given by the fact that the system is *live* and *lives* in an unpredictable environment: for such reason it is impossible to decide statically which implementation proves to be the most convenient. For instance, a static analysis might show that for any given input data size there exists an implementation which outperforms the others; yet many other factors (such as the variable system load) might prevent the static analysis to get real. For this reason the system must monitor the surrounding context and take decisions accordingly hence dynamically balancing all the constraints.

For self-aware adaptive computing system the ability to switch between different implementations of the same functionality while the system is running proves to be fundamental. The process through which this happens is called *hot-swap*. The hot-swapping of an implementation with another one is a non-trivial process; threads within a single process can access data structures concurrently and different implementations can use completely different data structures; state quiescence and state translation are the most visible problems the hot-swap process generates. As already described in Section 3.2.8, in self-adaptive computing literature this is a well-known problem and a general framework to solve it has already been proposed in [93]. This general framework inspired our hot-swap mechanism which is divided in 3 sub-phases as illustrated in Figure 4.2. These three sub-phases are:

a) a prior phase representing a common working condition;

b) a transfer phase in which new requests are blocked in order to reach a quiescent state which is translated to fit another data structure;

c) a final phase in which both blocked and new requests are allowed to proceed.

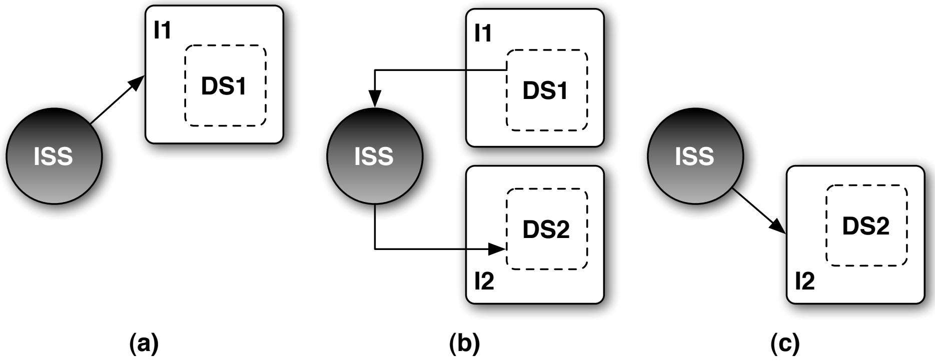


Figure 4.2: Hot-Swap mechanism

Even though such approach is inspired by the *state-of-the-art* it differs from it since it works solely on data structures instead of objects. This is due to the fact that not all implementations of a certain functionality must conform to a single interface. Furthermore, this approach defines a technology able to manage also the adaptation of the underlying physical architecture using, where possible, hardware implementations.

4.2.3 System overview

It is indeed true that many problems are more efficiently solved using hardware implementations instead of software implementations. Yet this claim actually depends on the expected QoS hence a software implementation might perform sufficiently well with respect to given constraints. Therefore two DES implementations were designed, one in software and one in hardware. The applications specify an expected performance goal over time (*e.g.,* 1000 blocks per second), the library translates the goal into an expected *heart rate* while the current heart rate is updated every time a block is computed. The *Application Heartbeats* makes it possible to check if the current heart rate fits the expected goal enabling the library to take decisions in accordance using an heuristic that avoids short-term oscillations. When the throughput (*i.e.,* heart rate) over a certain window of time drops under or excessively overcomes the expectations (and when other more subtle conditions are true) the library acts to improve performance. Since it is aware of the presence of two implementations the action consists in an hot-swap between them.

To avoid incurring in the state translation problem described in the previous section the library has been carefully designed in that the two DES implementations use the same underlying data structure. Moreover, since there is no need to change the underlying data structure the state quiescence problem is not raised and the library is not required to force the data structure into a stable state before the hot-swap is performed.

In more complicated applications the underlying data structure cannot be kept the same across different implementations. When this happens the state translation problem becomes real (*e.g.,* the translation from a list to a tree and vice versa is not straightforward and it strongly depends on how algorithms manages these data structures) and might become even worse if data structures are accessed by more than one thread concurrently since the state quiescence problem raises too.

The advantage of this library is that it hides complexity from the applications which are unaware of the presence of multiple implementations and are only required to set an expected performance goal.

4.2.4 `des-fpga` program

We have developed a stand-alone proof-of-concept application called `des-fpga` to evaluate our approach. Such program provides several command line options through which to control many of the underlying implementation parameters.

```
You can launch with no parameters: the program will run
indefinitely and encrypt/decrypt the same data over and over

Those are the possible parameters:
 —iter or –i: specify the number of iterations of the program
 —min or –m: minimum heartbeat rate
 —max or –M: maximum heartbeat rate
 —window or –w: window size
 —buffer or –b: buffer depth
 —swap or –s: minimum swap time among implementations
 —hbevery or –e: specifies the number of blocks after which an heartbeat is issued
 —allow or –a: allow multiple switches
 —nohwreconfig or –r: does not try to access hardware reconfiguration
 —triple or –t: use Triple DES
 —decrypt or –c: decrypt after encrypting
 —log or –l: prints heartbeat rate on screen
 —debug or –d: enables extensive logging
 —nohb or –z: disables heartbeats
 —onlyprinttime or –p: suppresses most prints (not execution time)
 —time or –x: measure execution time
 —version or –v: prints version and exits
 —help or –h: prints this help and exits
```

Listing 4.1: Embedded help of `des-fpga`

4.3 Services API

4.3.1 Functions

The *Services API* is made of several functions to initialize, register, receive commands, unregister, and shut down the service that makes use of them. In order to boost modularity of the system the functions are the same one for each possible service. Those are the cardinal ones:

`init_service()` This function takes care of the many details that have to be setup for the service before it can operate. It initializes all the data structures (in particular, `service_state` which hold the state of the service) and allocates the memory where the *Consensus Object* will write its commands.

`register_service()` It opens the default *named pipe* (currently `/tmp/consensus`) and write its registration details: name of the service, PID of the service, memory key to access the interface where the *Consensus Object* will write its commands, type of the service (*UP*, *DOWN*, or *UPDOWN*). This function is *blocking* in that until the *Consensus Object* does not start up and read from the pipe the `write` does not return.

`rename_service()` It allows to change the default name, which is "service-*PID*"

`read_command()` This function allows the service to check whether the *Consensus Object* has issued a new command. This function returns `true` or `false` depending on whether there is a new command or not. The commands coming from the *Consensus Object* are high-level commands since they are common to all services. A command can enable a service, disable it, or ask it to release resources. In the latter case a special technique is used to reduce the heart rate that the service sees.

`remove_all_dead_processes()` This function is used to constantly update the list of processes on which the service has effect. This is needed to avoid accessing a shared memory segment (the one where all the heart rate information is stored) that does not exist anymore (and that would probably lead to a crash) because the process has quit. This is very likely to happen since the service might still be accessing this portion before the *Consensus Object* manages to turns the service off on that particular target.

`unregister_service()` and `shut_down_service()` Those two functions are used to when the service is going to quit. Through the former function the *Consensus Object* removes the service from its internal list; through the latter function shared memory is cleaned up (which proves to be essential on systems such as Mac OS X where the default maximum shared memory size is particularly small).

4.3.2 Service guidelines

To adopt the *Services API*, a service has to correctly use a certain number of functions. This Section aims at guiding developers in build a service which is complaint to the proposed architecture:

- it is suggested that the service handles `ctrl+c` and, more in general, the `SIGTERM` and `SIGINT` signals. The signal handler should be used to set a flag (possibly of type `sig_atomic_t`, so that an atomic assignment is guaranteed); the main loop of the program should check it and, when set, exit the loop. This kind of procedure is necessary to correctly unregister the service (even if the *Consensus Object* constantly checks all registered services and processes to ensure that they have not quit without warning) and deallocate memory correctly

- the service should declare a variable of type `service_state_t` and initialize it
- the service should register with the apposite function
- the service should enter the main loop of execution. At each iteration the following actions have to be performed:
  - check whether there is a new command available
  - update the list of processes taking into account that some of them might have quit
  - obtain the lock to access information about the processes
  - for each targeted process obtain the heart rate information and enact the action policy typical of the service
  - release the lock
  - check whether the signal handler has been invoked
- when the service exits the loop (*i.e.,* , because the user has pressed `ctrl+c`) unregister the service and shut it down

The code for an "template" service is shown below. In the future we hope to make the functioning of the *Services API* more automatic.

```
/*
 *  Service example
 */

#include "services.h"
...
#define SERVICE_ACTION_INTERVAL 1000000
#define VERSION "1.0"


```

```
/* Global variable to stop execution */
/* It is volatile because gcc was performing
 weird optimizations (thanks Filippiello) */
volatile sig_atomic_t should_quit = 0;


/* Handles SIGINT adn SIGTERM (catches ctrl+c) */
void kill_handler(int signal_number) {
  should_quit = 1;
}


/* Main */
int main(int argc, char** argv)
{
  printf("Starting example service - version %s...\n", VERSION);
  boolean return_value = 0;

  /* Local variables */
  service_state_t  *service_state = (service_state_t *)malloc(sizeof(service_state_t));
  service_command_t *latest_command = NULL;
  process_service_record_t *current_process = NULL;
  double windowed_rate;
  double min_rate;
  double max_rate;
  double desired_rate;
  double global_rate;

  /* Catches ctrl+c and shuts down cleanly :) */
  struct sigaction sa;
  memset (&sa, 0, sizeof(sa));
  sa.sa_handler = &kill_handler;
  sigaction(SIGTERM, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);

  /* Initialization */
  return_value = init_service(service_state);
  if (return_value == false) {
    printf("Could not initialize service...\n");
  exit(-1);
}
else {
  printf("Service successfully initialized...\n");
}

/* Service-specific initialization */
...

/* Change service name */
rename_service(service_state, "service-example");

/* Register to the consensus object */
return_value = register_service(service_state);
if (return_value == false) {
  printf("Could not register service...\n");
  exit(-1);
}
else {
  printf("Service successfully registered...\n");
}

/* Main loop */
printf("Entering main loop of service...\n");
while(1)
{
  /*  */
  if(true==read_command(service_state, &latest_command))
  {
    /* Analyze command and possibly change behavior */
    /* Service specific code here */
    ...
  }

  /* Obtain lock because service is going to modify the shared segment */
  pthread_mutex_lock(&(service_state->
      interface->service_interface_mutex));

  /* Updates list of processes */
```

```
    remove_all_dead_processes(service_state);

    /* Act -for each process still alive */
    int n=0;
    for (n=0; n < service_state->interface->index ; n++) {
      current_process = &(service_state->interface->processes[n]);

      /* Skip dead processes */
      if (current_process->process_pid < 0) {
        continue;
      }

      /* Obtain heartbeats information */
      windowed_rate = hrm_get_windowed_rate(
          &(current_process->heartbeats_info) );
      min_rate = hrm_get_min_rate(
          &(current_process->heartbeats_info) );
      max_rate = hrm_get_max_rate(
          &(current_process->heartbeats_info) );
      global_rate = hrm_get_global_rate(
          &(current_process->heartbeats_info) );
      desired_rate = (max_rate + min_rate)/2;

      /* Set service specific values */
      ...


      if (windowed_rate <= max_rate && windowed_rate >= min_rate)
          {
        printf("Process %d is within expectations (m:%f w:%f
            M:%f, g:%f)\n",
              current_process->process_pid, min_rate,
              windowed_rate, max_rate, global_rate);
      }
      if (windowed_rate > max_rate) {
        printf("Process %d is above expectations (m:%f w:%f M:%f,
            g:%f)\n",
              current_process->process_pid, min_rate,
              windowed_rate, max_rate, global_rate);
      }
      else if(windowed_rate < min_rate) {
        printf("Process %d is below expectations (m:%f w:%f M:%f,
            g:%f)\n",
              current_process->process_pid, min_rate,
              windowed_rate, max_rate, global_rate);
      }

      /* Service specific action on process */
      ...

    }// end of for

    /* Release lock */
    pthread_mutex_unlock(&(service_state->interface->
        service_interface_mutex));

    /* Has user pressed ctrl+c ? */
    if(should_quit!=0)
    {
      printf("Quitting...\n");
      break;
    }

    /* If necessary, sleep before acting again */
    usleep(SERVICE_ACTION_INTERVAL);
  }

  /* Clean up and shut down */
  return_value += unregister_service(service_state);
  return_value += shut_down_service(service_state);
  if (return_value == true) {
    printf("Service has properly shutdown...\n");
  }
  else {
    printf("Service cleanup not completed... \n");
  }

  /* Service-specific cleanup */
  ...

```

```
  return 0;
}
```

Listing 4.2: Template for service development

4.3.3 Portability

The *Services API*, just like the *Heartbeats API*, has been compiled, run, and tested on Linux. As long as the service does not make use of any Linux-specific feature, the same code can be run also under Mac OS X.

4.4 Consensus Object

4.4.1 Functions

The *Consensus Object* API provides a certain number of function to handle service registration, to issue commands, and so on. Those are the most important functions:

`init_consensus_object()`

This functions initialize the data structure containing the status and creates the *named pipe* used for service registration.

`update_list_of_monitored_apps()`

This function handle programs registration. It checks whether new Heartbeat-capable applications have started up or whether previously existing applications have quite. This is performed checking the files in the `HEARTBEATS_ENABLED_ FOLDER`: in fact the *Application Heartbeats* create there one file per monitored application. The files are named after the PID of the process and are removed only if the application quits normally.

`update_list_of_services()`

This function checks whether any new service has registered or unregistered. This task is carried out reading from the *named pipe*.

`attach_consensus_object_to_service_memory()`

This function attaches the *Consensus Object* to the shared memory segment allocated by a service. Such segment is where commands are written.

`dump_list_of_services_to_file()` and **`read_list_of_services_from_file`**

These functions are used to periodically save the list of services to file. This is needed since when the *Consensus Object* reads from the *named pipe* and registers a service, the only trace for the registration is in the *Consensus Object*'s memory. If the *Consensus Object* quits or crashes, after the following start up, it will not know which services are available. For this reason it has to save the list to file and, at start up, check whether those services still exist. This procedure is not necessary for applications because their registration is not handled through a *named pipe* but through files (which are persistent).

`shut_down_consensus_object()`

Cleanly shuts down the *Consensus Object* by deallocating resources.

`enable_service(),disable_service(), release_resources_faking_heart_rate(),` and **`make_service_overperform()`**

Those are the possible commands that the *Consensus Object* can issue. The *Consensus Object* specifies whether a service should be enabled (first function), disabled (second function), asked for resources (third function), or asked to make the target overperform (fourth function).

Since most services can target one application at a time, all these functions also require a target parameter.

`get_service_state_for_process()`
This function can be used to check whether a service is active on a given process.

4.4.2 Consensus Object guidelines

Thanks to the provided API it is possible to easily re-implement the *Consensus Object*. Those are the guidelines:

- allocate a data structure of type `consensus_state_t` and use the given initialization function before starting
- periodically check whether new services have registered/unregistered (the API automatically attaches the segment of the new services or detaches the segment of an old service)
- periodically check whether new processes have registered/unregistered
- periodically check that registered processes and services are still functioning and have not crashed
- given the information about a process performance decide which service to enable or disable
- at the end of execution call the function to release resources

4.4.3 Portability

The *Consensus Object API*, just like the *Heartbeats API*, has been compiled, run, and tested on Linux and Mac OS X. It will work on Macs as long as services do not make use of Linux-specific features.

4.4.4 Decision engine

The pseudo-code shown in Listing 3.1 is implemented as shown below.

```
/*** Functions ***/
/*** Reinforcement learning Function prototypes ***/

/* Produces binary representation (in a string) of a byte */
const char *byte_to_binary (int x);

/* Prints binary representation of an array of char; for each
 * char the binary representation is shown; optionally a character
 * can be passed and will be printed before the representation */
void print_binary_state_with_char(char* array, char c);

/* As above but with no option for printing a char before the
 * binary representation */
void print_binary_state(char* array);

/* Set to one a BIT in the state matrix */
void set_to_one(unsigned char* array, unsigned int row, unsigned
    int column);

/* Set to zero a BIT in the state matrix */
void set_to_zero(unsigned char* array, unsigned int row, unsigned
    int column);

/* Backs up state matrix, sets to zero a bit in the copy and
 * returns the copy */
unsigned char* copy_and_set_to_zero(unsigned char *matrix,
    unsigned int row, unsigned int column);

/* Backs up state matrix, sets to one a bit in the copy and
 * returns the copy */
unsigned char* copy_and_set_to_one(unsigned char *matrix,
    unsigned int row, unsigned int column);
```

```

/* Set an entire column of the state matrix to one
 * Useful during initialization */
void set_column_to_one(unsigned char* array, unsigned int column);

/* Retrieves from the state matrix which relationship links a
    service
 * (specified as a column) and a process (specified as a row).
 * For instance "ON", if the service is active on that process */
command_t get_current_command(unsigned char* array, unsigned int
    row, unsigned int column);

/* Takes the apps_performance array (part of the state) and build
 * an alternative representation using an int. It is necessary
 * to create a key for the hash table */
unsigned int convert_to_int_key(unsigned short int *array);

/* Prints the Q matrix with a char before each line
 * The Q matrix holds the state-action value */
void print_q_matrix_with_char(double* array, char c);

/* Prints the Q matrix as above but no option to
 * print a char before each line */
void print_q_matrix(double* array);

/* Function that iterates on the state hash table */
void state_iterator(gpointer key, gpointer value, gpointer
    user_data);

/* Allocates and return a new Q matrix (all zeros) */
double* malloc_q_matrix();

/* Backs up the current Q matrix, modifies it and then
 * return the copy */
double* copy_and_update_value(double *matrix, unsigned int row,
                              unsigned int column, double value);

/* Function to iterate on the performance has table */
void performance_iterator(gpointer key, gpointer value, gpointer
    user_data);
```

```

/* Print the complete state (performance hash table, state
 * hash table and Q matrixes) */
void print_complete_state(GHashTable* performance_hash);

/* Print state matrix and linked Q matrix */
void print_state_and_q(GHashTable* state_hash);

/* Updates apps performance given current heart rates */
void update_apps_performance(unsigned int* apps_performance,
                             process_record_t *current_process);

/* Allocates a new state matrix properly initalized */
unsigned char* malloc_state_matrix();

/* Find maximum value in a Q matrix and return a pointer to it */
double * find_max_q(double * matrix);

/* Given a state (as performance hash plus state matrix)
 * retrieve the Q matrix and return the maximum value
 * contained in it (NOT a pointer to the value!) */
double find_max_q_given_state(GHashTable* performance_hash,
                              unsigned int* apps_performance,
                              unsigned char* state_matrix);

/* Sum all elements of a Q matrix and return the total */
double sum_of_all_qs(double* matrix);

/* Backups a Q matrix, normalize it with respect to the total
 * sum of its elements (see previous function) and return it */
double* normalize_matrix(double *matrix);

/* Choose action using the SOFTMAX method */
double* softmax_action_selection( double* matrix, unsigned int *r, unsigned int *c );

/* Performs the first part of the R-learning algorithm,
 * choosing a proper action, updating some values and returning
 * a number of variable: an updated state matrix, a command to
 * be sent to the consensus object (please note that PID
    translation
 * is required), a pointer to the maximum values of Q and a pointer
 * to the Q of the chosen action */
char* select_action(GHashTable* performance_hash,
                    unsigned short int* apps_performance,
                    unsigned char* state_matrix,
                    double** old_max_q, double** chosen_q,
                    service_command_extended_t* command);

/* This function define the reinforcement given by the environment
 * It follows a simple heuristic */
unsigned int get_reinforcement(unsigned int* apps_performance);

/* Given the index of a service this function returns the
 * PID of the service */
service_record_t* give_pointer_to_service_number(pid_t serv, service_record_t* s);

/* Given the index of a process this function returns the
 * PID of the process */
pid_t give_pid_of_app_number(pid_t target, process_record_t* p);

unsigned int count_processes(process_record_t* processes);
unsigned int count_services(service_record_t* services);

/* ... */

int main() {

  /* ... */

  while(...) {

    if(...) {
      select_action(performance_hash, apps_performance,
      state_matrix, &old_max_q, &chosen_q, &command);

      /* Convert indexes to PID */
      service_record_t* temp_service = NULL;
      temp_service =
          give_pointer_to_service_number(command.service,
          consensus_state->list_of_services);
      command.target = give_pid_of_app_number(command.target,
          consensus_state->list_of_processes);

      if(command.command == ON)
        enable_service(temp_service, command.target);
      else
        disable_service(temp_service, command.target);

      usleep(CONTROL_ACTION_INTERVAL);

      update_apps_performance(apps_performance,
          consensus_state->list_of_processes);

      reward = get_reinforcement(apps_performance);
    }
  }

  /* ... */
}
```

Listing 4.3: Source code (simplified) for the R-learning algorithm

Chapter 5

Experimental results

"No amount of experimentation can ever prove me right;
a single experiment can prove me wrong."
Albert Einstein

"A state-of-the-art calculation requires
100 hours of CPU time on the state-of-the-art
computer, independent of the decade."
Edward Teller

In this Chapter I present the experimental results that support the validity of the use of the *Application Heartbeats* and the *Consensus Object* in a real implementation. In particular this Chapter first analyzes the overhead of the *Application Heartbeats* on the execution time of two example applications (Section 5.1); it then presents the results linked to the hardware/software implementation of an *Implementations Library*; finally some *Consensus Object* and *Services API* tests outcomes are shown.

5.1 Application Heartbeats overhead with `des-fpga`

In this Section the analysis of the overhead on execution time of applications adopting the *Heartbeats* framework is presented. Such investigation is extremely important in the context of autonomic systems since the monitoring infrastructure must be as lightweight as possible: as already presented in Figure 2.1.3 the performance gain given by the implemented self-properties cannot be nullified by the necessary monitoring architecture.
Such overhead is due to a series of system calls that the Heartbeats framework makes in order to initialize its data structures and update the global heart rate. Moreover, the overhead seems to decrease while the number of blocks increases due to the fact that the weight of the initialization tends to decrease. The initialization of the Heartbeats make use of `getpid()`, `shmget()`, and `shmat()` while the global heart rate is updated by means of two calls to `clock_gettime()` inside the `heartbeat()` call.
Two examples are presented: both execute the same code and are run under a Linux-based Operating System but while one executes on a *Xilinx Virtex-II Pro FPGA board* (more details can be found in Section 5.2.1) the other runs on a common *x86-based* computer.
The chosen application is a software implementation of the *DES* cryptographic algorithm [96]. While there are many other algorithms that would have served the same purpose and that would have reasonably brought to similar results, DES has a relatively simple implementation and requires the simplest possible input: text.
To benchmark the impact of the *Heartbeats* on the execution time I developed a C program that implemented the DES algorithm and that could be instructed not to use the *Application Heartbeats*. The application is called `des-fpga` and it starts a timer just before encrypting the first block and stops it right after encrypting the last one. The experiment has been organized this way:

- almost 30 different input sizes were tested in the range 1 to 1000 blocks.
- in order to balance slow downs and fortuities on the system the executions were repeated 10 times and then averaged
- the execution times were compared and the average percentage impact was calculated

5.1.1 Benchmarks on a FPGA-based system

Table 5.1 shows a sample of the data gather for the FPGA-based solution while Table 5.2 shows the averaged values that have been used to plot the graph shown in Figure 5.1.
The average impact on the execution time is 3.52%, which is both moderate and seems to become even lower the greater the number of blocks.

5.1.2 Benchmarks on a x86-based system

After running the same test on an x86-based computer (Intel Core 2 Duo running at 2.4GHz, 2MB L2 cache, 4 GB RAM) an extremely puzzling result is obtained: not only the overhead is higher, but it is circa 66% of the execution time. At first glance the *Heartbeats* seem to have performed well in the FPGA-based scenario just because their overhead was "hidden" by a slow CPU. The truth is quite different, though. In `des-fpga` by default issues one heartbeat per encrypted block. Considering that one DES block is made of 64 bits (8 Bytes) and that on the x86 machine used for the experiment its encryption takes less that 1μs, the overhead of system calls becomes huge. This happens because system calls can be considered as a fixed cost to be paid and issuing thousands of heartbeats per second both defeats the pur-

	Number of encrypted blocks								
	1		2		3		4		…
Execution time (µs)	604	369	641	634	832	1036	1264	977	…
	379	371	646	623	847	838	1002	966	…
	376	600	679	887	842	818	1245	1198	…
	628	367	641	631	840	815	999	965	…
	407	373	847	627	835	821	1017	1208	…
	381	374	642	625	844	807	1007	981	…
	384	375	640	626	1020	817	1001	975	…
	378	564	655	642	861	812	1234	1201	…
	378	374	640	619	1050	817	995	991	…
	384	376	667	623	836	811	1013	994	…
Average (µs)	429.9	414.3	669.8	653.7	880.7	839.2	1077.7	1045.6	…
Impact (%)	3.6287		2.4037		4.7121		2.9785		…

Table 5.1: Table showing a sample of the gathered data for the FPGA-based example

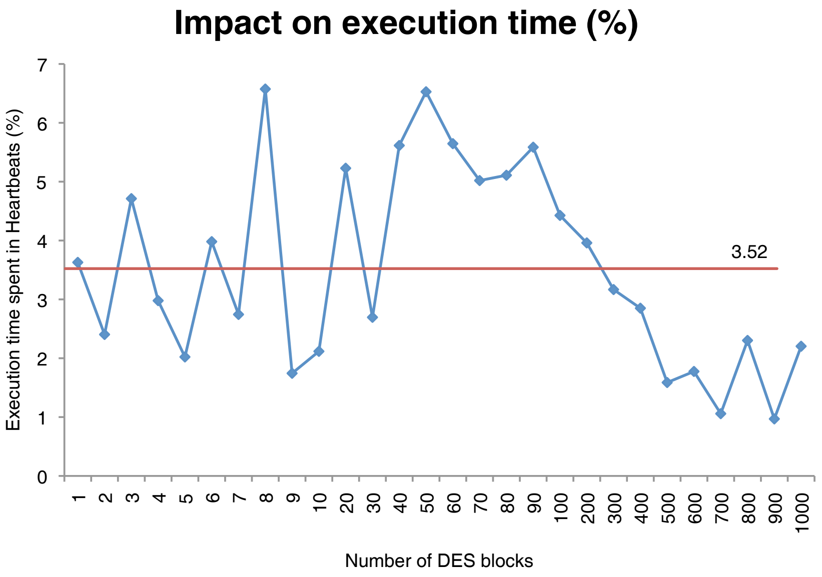


Figure 5.1: Application Heartbeats API overhead in the FPGA-based scenario

# blocks	1	2	3	4	5	6	7	8	9	10
Average impact (%)	3.628	2.403	4.712	2.978	2.021	3.981	2.743	6.574	1.743	2.116
Average execution time (µs)	430	670	881	1078	1251	1447	1603	1859	1967	2083

# blocks	20	30	40	50	60	70	80	90	100	200
Average impact (%)	5.228	2.695	5.614	6.528	5.646	5.019	5.109	5.585	4.426	3.959
Average execution time (µs)	4183	6244	8591	11078	13641	16290	19065	22304	25400	64240

# blocks	300	400	500	600	700	800	900	1000	10000	100000
Average impact (%)	3.166	2.850	1.587	1.775	1.057	2.302	0.969	2.203	9.123	13.350
Average execution time (ms)	117.5	185.6	267.1	365.3	476.3	604.6	744.8	908.6	2688.2	20822.9

Table 5.2: Table showing the average execution time and the percentage average impact on it caused by the the *Application Heartbeats* in the FPGA-based scenario

pose of the framework and generates a huge amount of system calls. Obviously, since the CPU was much slower, this did not happen in the FPGA-based scenario.

Taking into account these considerations it has been necessary to slightly reduce the number of heartbeats. For this experiment heartbeats were issued every 256 blocks: this quantity seems particularly reasonable since it is a multiple of the usual smallest dimension of files under Linux.

The results obtained encrypting a number of blocks multiple of 256 are extremely similar to the ones obtained before. The experiment has been organized this way:

- 15 different input sizes were tested in the range 256 to 102400 blocks.
- in order to balance slow downs and fortuities on the system the executions were repeated 10 times and then averaged
- the execution times were compared and the average percentage impact was calculated

Sample data is shown in Table 5.3, while final results are displayed in Table 5.4 and Figure 5.2.

5.2 Implementations Library: putting DES to test

In this Section we put to test the *Implementations Library* that has been described in Sections 3.2.8 and 4.2.

5.2.1 Environment

For the implementation a Xilinx University Program Virtex-II Pro (XUPV2P) board featuring an *XC2VP30* FPGA used to provide the basic architecture

	Number of encrypted blocks								
	256		512		768		1024		…
Execution time (µs)	189	187	352	341	519	502	676	717	…
	189	180	347	340	513	501	675	661	…
	189	180	352	342	519	543	671	684	…
	234	277	364	344	514	502	675	684	…
	191	180	352	380	516	523	693	663	…
	189	185	351	341	731	503	671	682	…
	222	180	399	343	511	508	674	662	…
	189	180	351	340	507	505	675	662	…
	188	198	356	341	516	502	832	663	…
	191	180	365	343	513	504	675	662	…
Average (µs)	197.1	192.7	358.9	345.5	535.9	509.3	691.7	674	…
Impact (%)	2.2323		3.7336		4.9636		2.5589		…

Table 5.3: Table showing a sample of the gathered data

# blocks	256	512	768	1024	2048	3072	4096	
Average impact (%)	2.2323	3.7336	4.9636	2.5589	0.9450	3.5766	4.5273	
Average execution time (µs)	197.1	358.9	535.9	691.7	1396.7	2055	2809.6	
# blocks	5120	6144	7168	8192	9216	10240	51200	102400
Average impact (%)	5.0312	1.2884	7.0051	6.0787	1.6619	6.6600	4.6812	3.9150
Average execution time (µs)	4501.9	4742	5989.9	8347.1	6576.7	8168.1	44230.1	711086.1

Table 5.4: Table showing the average execution time and the percentage average impact on it caused by the the *Application Heartbeats* in the x86-based scenario

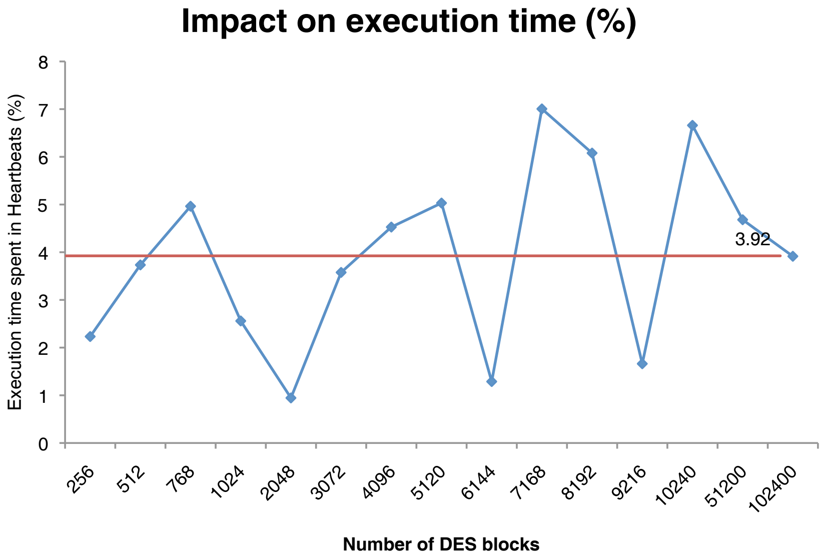


Figure 5.2: Application Heartbeats API overhead in the x86-based scenario

and I/O ports to run and interact with the operating system. The FPGA comes with an integrated hard processor, a *PowerPC 405* with a maximum frequency of 300 MHz with 16 kB of instruction cache and 16 kB of data cache; the PowerPC 405 is used as the General Purpose Processor (GPP) of our embedded system and is allowed to access 256 MB of DDR-266 used as system memory through the *Xilinx Multi-Port Memory Controller (MPMC)* Double Data Rate (DDR) memory controller. On top of this architecture we loaded the *Linux vanilla-kernel* version 2.6.33.1 and a reduced root file system built upon *busybox* version 1.16.0 and both the kernel and the root file system are compiled with a toolchain featuring *gcc* version 4.2.4 and *uClibc* version 0.9.30.1. When the board is powered-on the bitstream implementing the architecture is taken from the first partition of the onboard flash memory and used to program the FPGA while the kernel and the root file system are taken from the second partition of the same flash memory. The flash memory is accessible thanks to *Xilinx SystemACE* storage controller. The console of the kernel is redirected to the *Xilinx UARTLite* serial controller and the embedded system is controller by means a *Telnet* session passing through the *Xilinx Ethernet Lite* network interface controller. The architecture contains also an IP-Core with a *Wishbone* interface implementing DES encryption and decryption capabilities and is connected to the rest of the system by means of an IP-Core implementing a Processor Local Bus (PLB) version 4.6 to Wishbone bridge. A schematics of the architecture is proposed in Figure 5.3. In a real scenario the FPGA might need to be reconfigured before having the Core DES available.

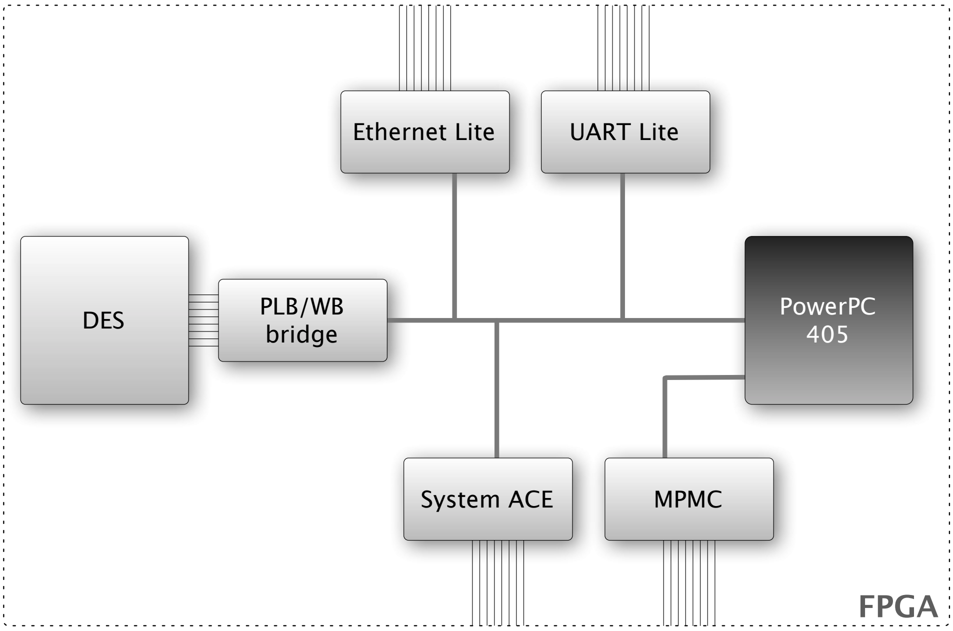


Figure 5.3: FPGA architecture

5.2.2 Static analysis

The first result of our work is to statically analyze the ideal behavior of the two DES implementations. We run both the implementations varying the input data size and averaging the results on tens of different trials. The results are shown in Figure 5.4: the hardware implementation (*square*) is faster than the software implementation (*circle*) when the input size is bigger than 50 blocks. The hardware implementation might need to be configured hence the hardware implementation considering the reconfiguration time (*triangles*) gets faster than the software implementation when the input size is bigger than 400 blocks.

This analysis might seem to prove that, depending on the input size (number of blocks), it is possible to choose the best implementation statically. Yet in a dynamic scenario such execution times might change completely due to the system load or to runtime constraints (*e.g.*, power consumption): in this case a statical approach might fail. Since many other factors cannot be predicted it is necessary to monitor the throughput of the active implementation and decide at runtime when to switch one in favor of the other.

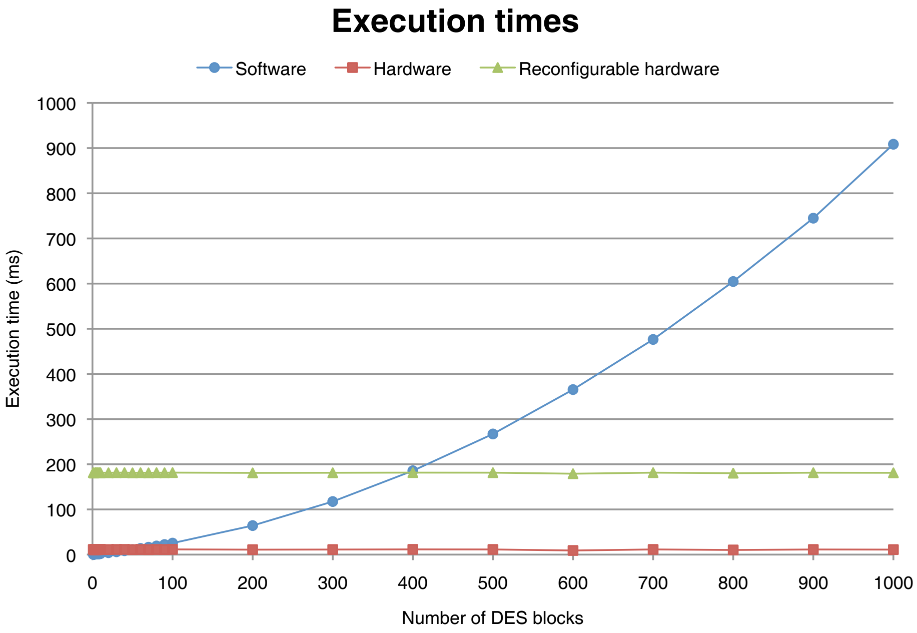


Figure 5.4: Execution times

5.2.3 A dynamic scenario

Figure 5.5 shows a execution scenario of the *Implementations Library*. Even though in (t_0, t_1) the application heart rate is dropping due to the context switches we are not observing any change in the implementation because the current heart rate is still inside the desired heart rate window. In (t_1, t_2) the computed heart rate exits this window for more than Δ time instants hence the ISS decides to spend (t_2, t_3) reconfiguring the FPGA and to switch to the hardware implementation. The Δ time depends on the decision mechanism used within the system while the reconfiguration time R is spent only when the desired hardware implementation is not already configured on the FPGA. In (t_3, t_4) the heart rate increases entering the desired heart rate window. The sudden decrease of the heart rate in (t_4, t_5) proceeding in (t_5, t_6) is caused by resources contention (*e.g.*, buses, memory, etc...). Therefore, after waiting for Δ time instants the software implementation is swapped in to try to increase the heart rate as it happens in (t_6, t_7).

The execution scenario of the *DES Implementations Library* stands as an instance of *enabling technology* in the field of autonomic computing: the proposed methodology is valid since the information gathered at runtime enables the system to react to unpredictable and unknown conditions.

Even if at present it is not possible to guarantee in any situation a fulfillment of the *QoS*, the *Implementations Library* here introduced brings a clear improvement over static implementations.

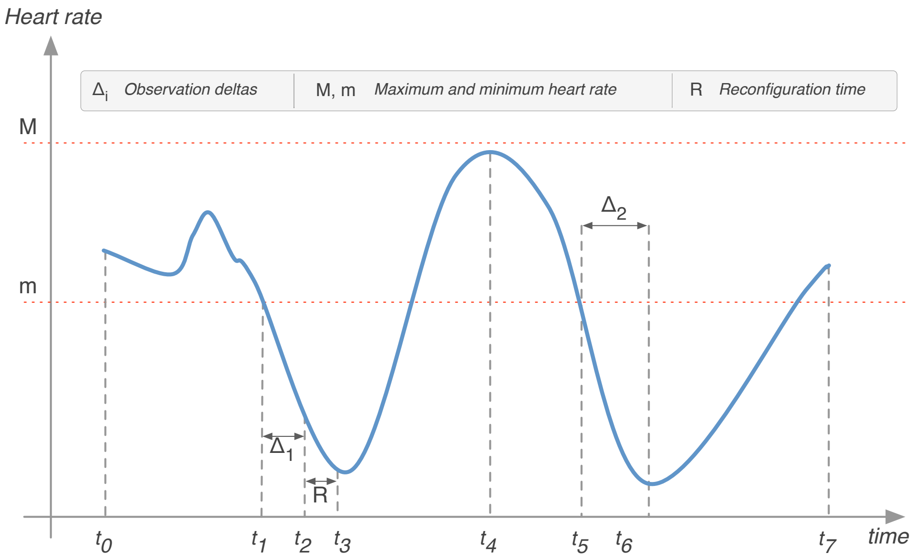


Figure 5.5: Application execution behavior. The interval between m and M defines the desired heart rate window

5.3 Consensus object and Services

5.3.1 Testing the overhead of the framework

The purpose of the following test is to quantify the overhead brought by the infrastructure designed and implemented in the previous Chapters. As we have seen throughout this dissertation, self-adaptive systems bring tremendous advantages over modern computing system. Of course we do not want this advantages to vanish because of overhead: the infrastructure must be lightweight.

We have proven in Section 5.1 that the *Application Heartbeats* bring a low overhead. It is now the time to analyze the *Consensus Object* and the *Services API*: we first analyze the time needed to initialize, register a service and read a command (Section 5.3.1); we then measure the time needed to register a process and write a command (Section 5.3.1); as a final step we analyze an implemented service and we compare performance of the

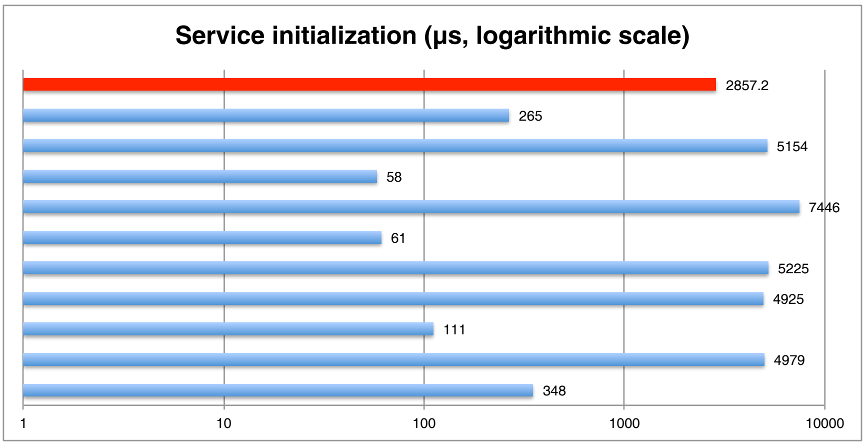


Figure 5.6: Service initialization. Ten executions, average is in red.

stand-alone version against the *Consensus Object*-driven one (Section 5.3.1). The test platform is the same one used in Section 5.1.2 and consists of an Intel Core 2 Duo running at 2.4GHz, 2MB L2 cache, and 4 GB RAM.
The used methodology is the same across all tests: the same experiment has been repeated ten times and then averaged.

Services overhead tests

In this series of tests we analyze the absolute time required to initialize a service, register it to the *Consensus Object*, and read a command from the *Consensus Object*.
Initialization (Figure 5.6) is shown on a logarithmic scale because it takes an extremely variable amount of time. Debugging the few lines of code that are part of the initialization function, the "culprit" immediately stands out: `ftok()`. `ftok()` is the function that has to generate a unique memory key to be used to address the shared memory segment. It takes a variable amount of time to execute and, sometimes, the initialization function has to call it repeatedly (up to a maximum of three times) before it successfully

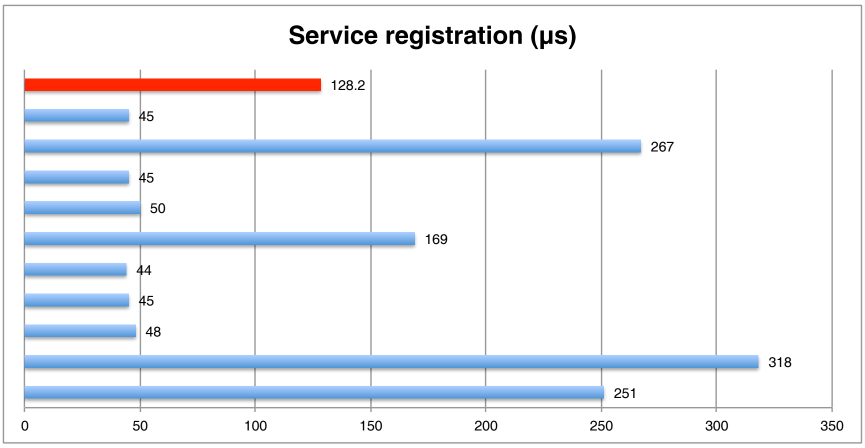


Figure 5.7: Service registration. Ten executions, average is in red.

returns a valid memory key.

Registration (Figure 5.7) is much easier to explicate (please also note that it is not on a logarithmic scale) since the time to write to the *named pipe* can easily vary.

Reading a command coming from the *Consensus Object* (Figure 5.8) requires the service to obtain an exclusive lock, read the shared memory segment, release the lock, and update its internal structures.

Consensus Object overhead tests

Process registration (Figure 5.9) is performed by the *Consensus Object*, which scans for files in the *HEARTBEAT_ENABLED_DIR*, considers only files entirely made of numbers, and adds or removes processes from its list. It is easy too see that access to the file system has a certain degree of unpredictability.

Writing a command (Figure 5.10) has proved very efficient and the time to perform such action is extremely stable. This happens only because the driven service only obtains the exclusive lock when necessary and releases

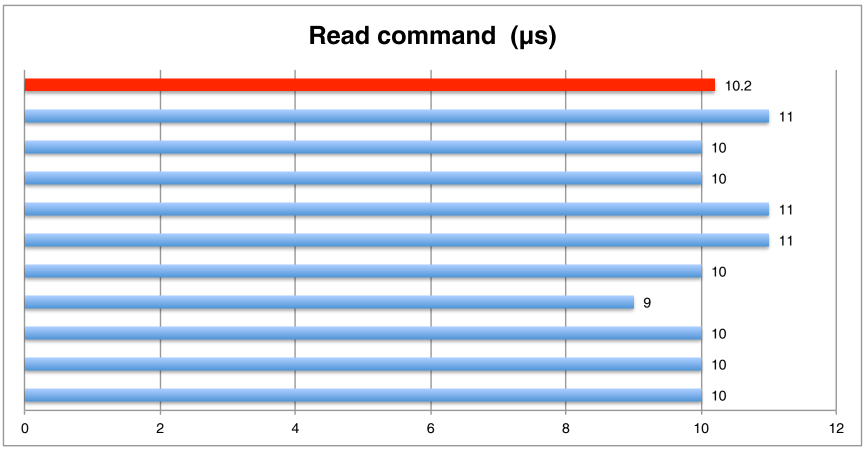


Figure 5.8: Reading a command. Ten executions, average is in red.

it right after: if the service were less efficient and less optimized (and would then keep the lock much longer) the time to write a command would dramatically increase.

The reason why writing a command requires less time that reading one is that when a service reads a command it also has to update its internal structures.

Nevertheless *shared memory* proves to be an extremely fast for of Inter-Process Communication (IPC).

Overhead on core allocator service

Summarizing the results obtained in Section 5.3.1 it is possible to say that the centralized autonomic infrastructure has an average overhead of 2996µs, which is almost all due to initialization. Please note that this is a non recurring allowance (it is only required at service startup) and it can be considered negligible.

We now analyze the overhead (Figure 5.12 and Figure 5.13) brought by the proposed *Consensus Object*-driven approach on the execution of an imple-

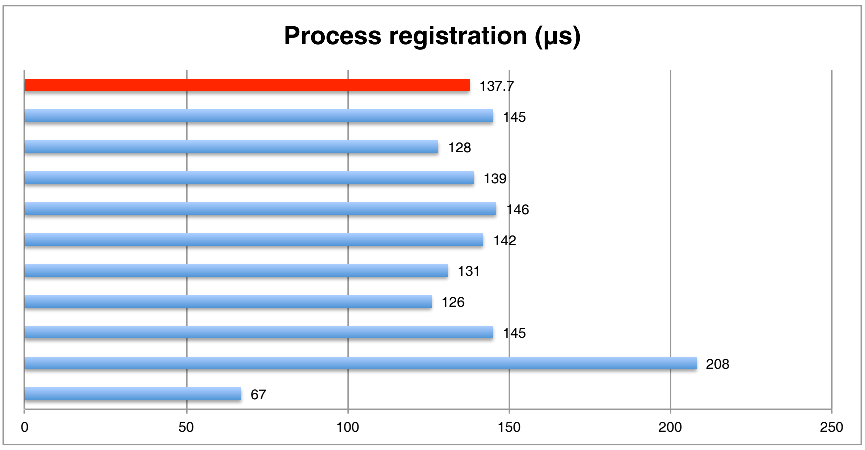


Figure 5.9: Process registration. Ten executions, average is in red.

mented service: the *core allocator*. We have developed two versions:

- a stand-alone version that automatically starts monitoring one Heartbeat-enabled application and enacts one of its policy of core allocation

- a *Consensus Object*-driven one that initializes, registers, and reads commands using the *Services API*; when enabled it enacts the same policy as the other version

The results over ten executions are quite stable and clear: the overhead on the maximum possible throughput of the service is 15.81%. Such result is encouraging since the *core allocator* constantly reviews and modifies its action; many other services "sleep" for a few microseconds before starting to consider changes in their policy. This means that on many services the overhead will be much less than 15.81%.

5.3.2 Machine learning-based decision engine

As explained in Section 3.3.2 and 4.4.4 the decision engine of the *Consensus Object* is based on machine learning and, in particular, on the rein-

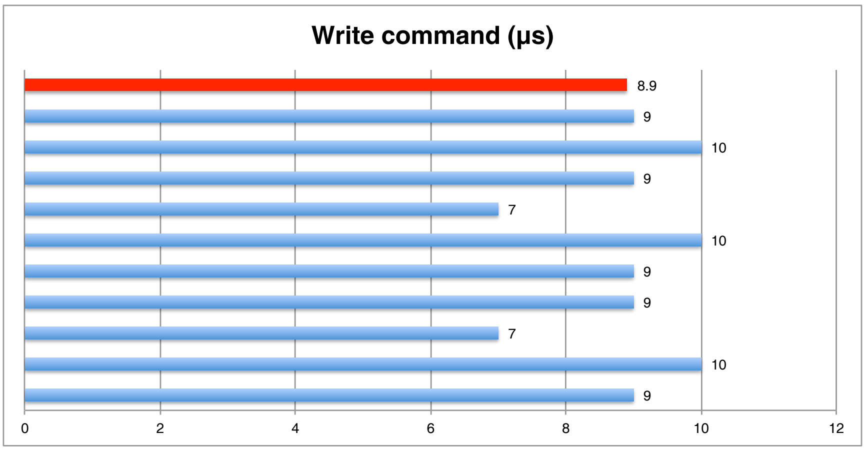


Figure 5.10: Writing a command. Ten executions, average is in red.

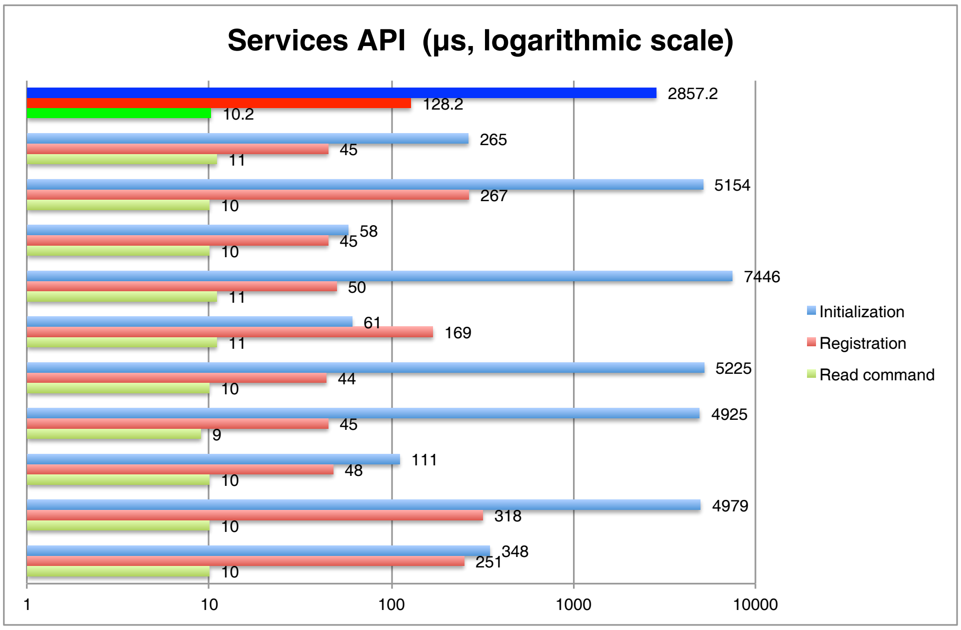


Figure 5.11: Overhead of the centralized autonomic infrastructure at service startup. Average values are shown at the top.

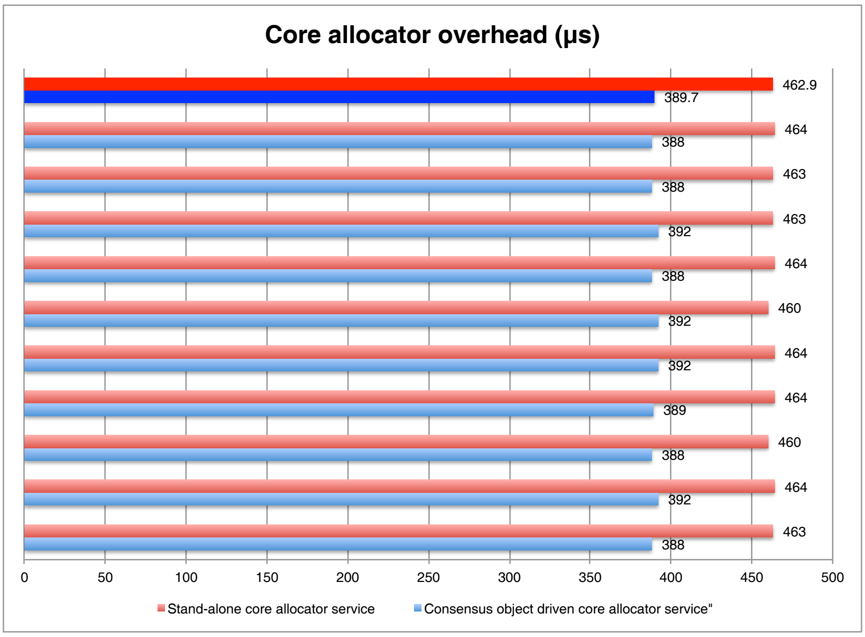


Figure 5.12: Service

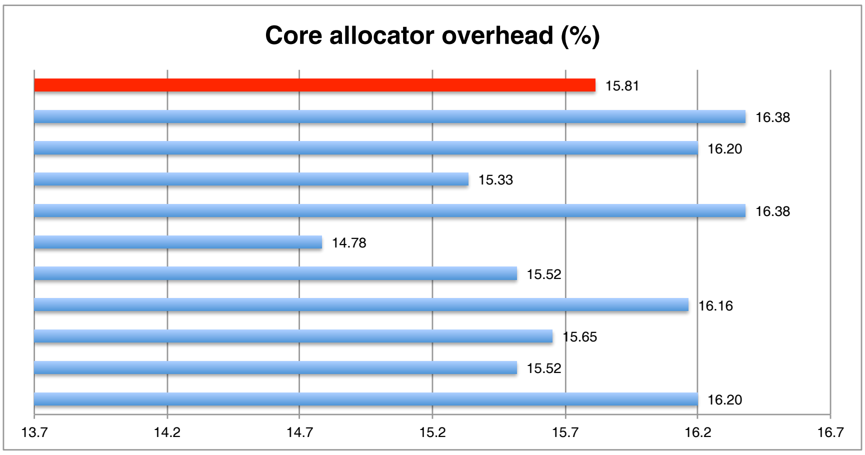


Figure 5.13: Core allocator: overhead of the centralized autonomic infrastructure. The average value is shown in red.

forcement learning algorithm called *R-learning*. Without any doubt it is the most delicate part of the whole framework since it is responsible for the activation and deactivation of services.
The implemented algorithm makes use of *Softmax* for the choice of actions; such method is not deterministic and it makes the whole behavior non deterministically definable. For this reason we have repeated our tests several times and we now show how reliable the decision engine is.
We present seven different tests, which have been carried out:

- on an x86-64 machine featuring an Intel Core i7-870 processor (2.93 GHz) used as GPP, 4 GB of SDRAM DDR3-1333 of main memory, and an NVIDIA GeForce GT 240 graphic card used as Graphics Processing Unit (GPU)
- using the *core-allocator* multi-application service
- running one or more instances of the `x264` program, possibly with different parameters, a different number of threads, and different heart rate goals

Free execution

This first test has been carried out to explore the heartbeats' generation by the `x264` program and see whether it has major cycles or major variations. As shown in Figure 5.14 the heart rate is almost constant. Its average, with the given parameters and the given number of cores, is 85.49 heartbeats per second.

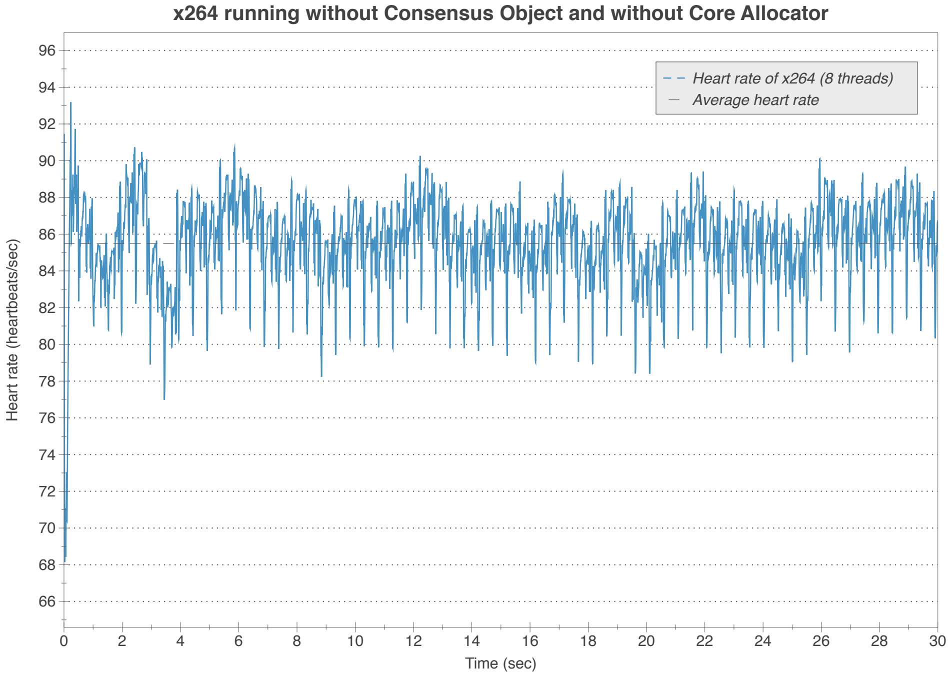


Figure 5.14: Heart rate of an instance of `x264`. Neither the *Consensus Object* nor the core allocator service are running.

Learning to enable a service

This scenario shows x264 running with the *core allocator service* controlled by the *Consensus Object*. x264 states a minimum of 20 and a maximum of 30 hearbeats per second. In the first seconds of the execution the heart rate increases since the *Consensus Object* is still learning to maximize the reinforcement. After 5-6 seconds circa the *Consensus Object* finds the actions that maximize the reinforcement (and that bring the heart rate in the required range). Such actions are:

- enables the *core allocator* service whenever above the stated performance goals
- do not disable the *core allocator* whenever the performances are between the minimum and maximum heart rate

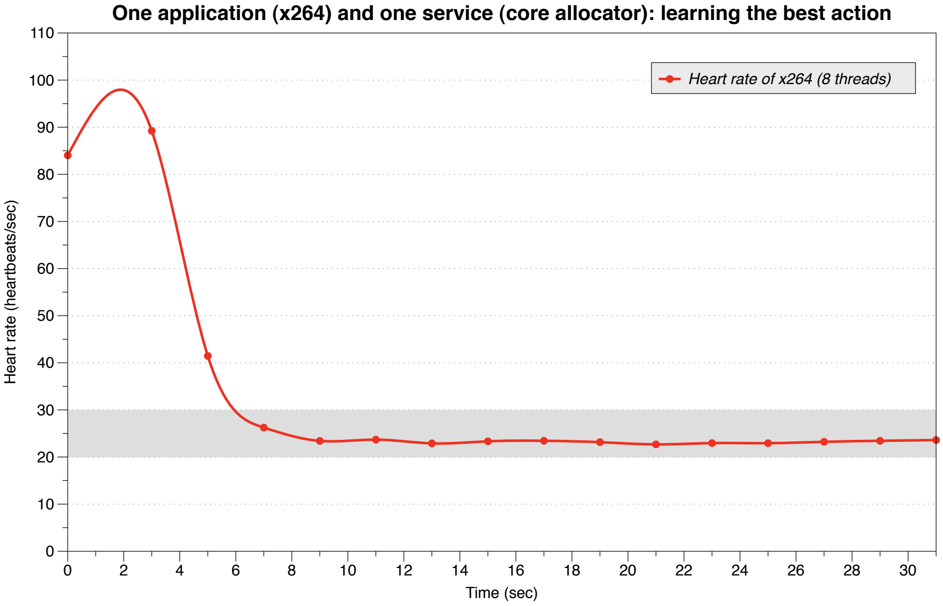


Figure 5.15: The *Consensus Object* learns to use a service when it is useful to achieve the given performance goal. Desired heart rate range is in gray.

Learning to disable a service

This scenario shows again one instance of `x264` running with an ad-hoc service controlled by the *Consensus Object*. `x264` states a minimum of 35 and a maximum of 50 heartbeats per second. In the first seconds of the execution the heart rate decreases since the *Consensus Object* is still learning how the service alters the heart rate. After 5-6 seconds circa the *Consensus Object* finds the action that maximizes the reinforcement (and that brings the heart rate in the required range), which is to disable the service whenever the heart rate is below the required performance.

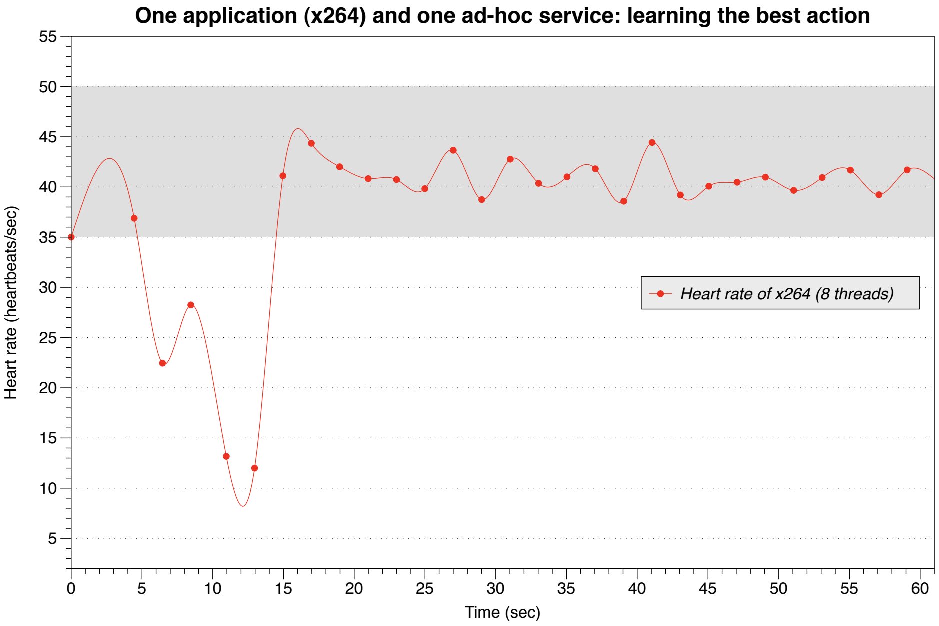


Figure 5.16: The *Consensus Object* learns to disable services that are not helping in achieving the performance goals. Desired heart rate range is in gray.

Adaptation to new runtime conditions: two applications with the same performance goal

In this test we have analyzed the reaction of the system to the entry of a new application. As shown in Figure 5.17, the decision engine first learns how to handle one application and it then understands how to keep both applications in their performance range. The heart rate oscillates a little bit because of the way the *core allocator* decides how many cores to assign to each process. Such behavior can certainly be improved and it is a hint for future works.

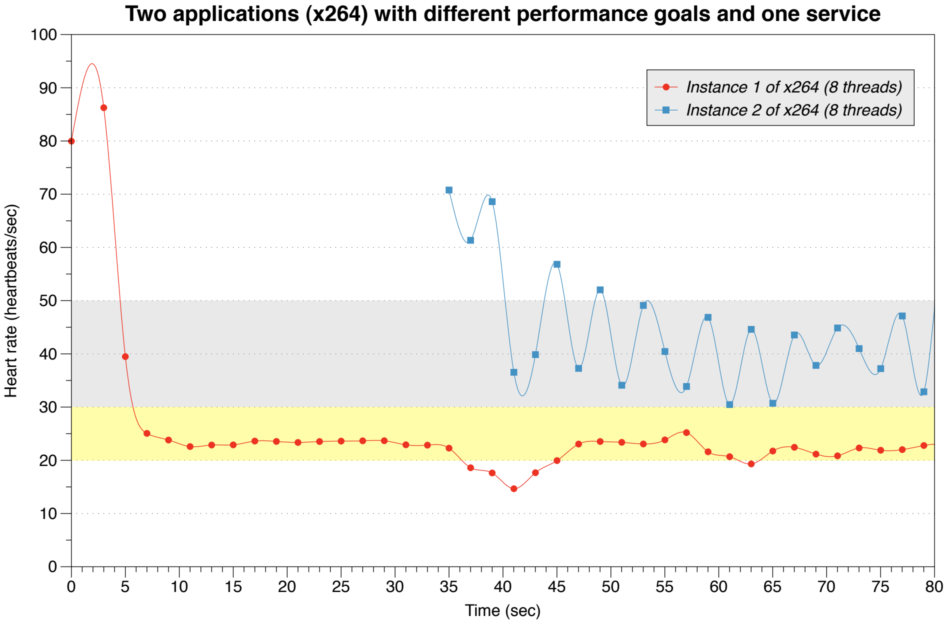


Figure 5.17: Two instances of x264 with different performance goals. Desired heart rate ranges are the yellow (for the first instance) and gray (for the second instance) bands.

Adaptation to new runtime conditions: two applications with different performance goals

This test verifies that the behavior highlighted in the previous test is still valid when the performance goals for the two instances of x264 are different.

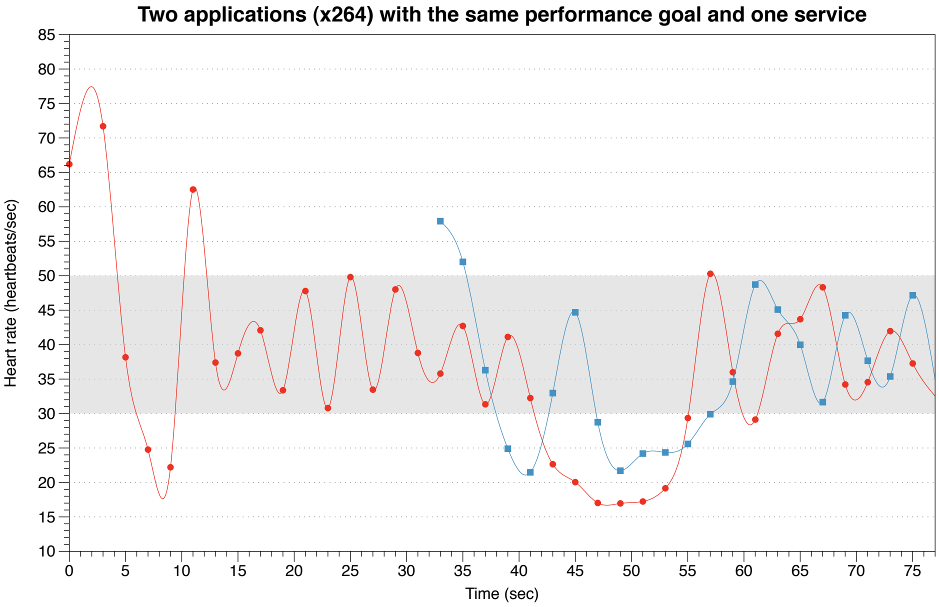


Figure 5.18: Two instances of x264 with the same performance goals. Desired heart rate range is in gray.

Adaptation to new runtime conditions: four applications

In this scenario four instances of x264 are started five seconds one after the other. With respect to the previous test, the solution space is much bigger. Yet the heart rate of the four applications converges in a matter of seconds.

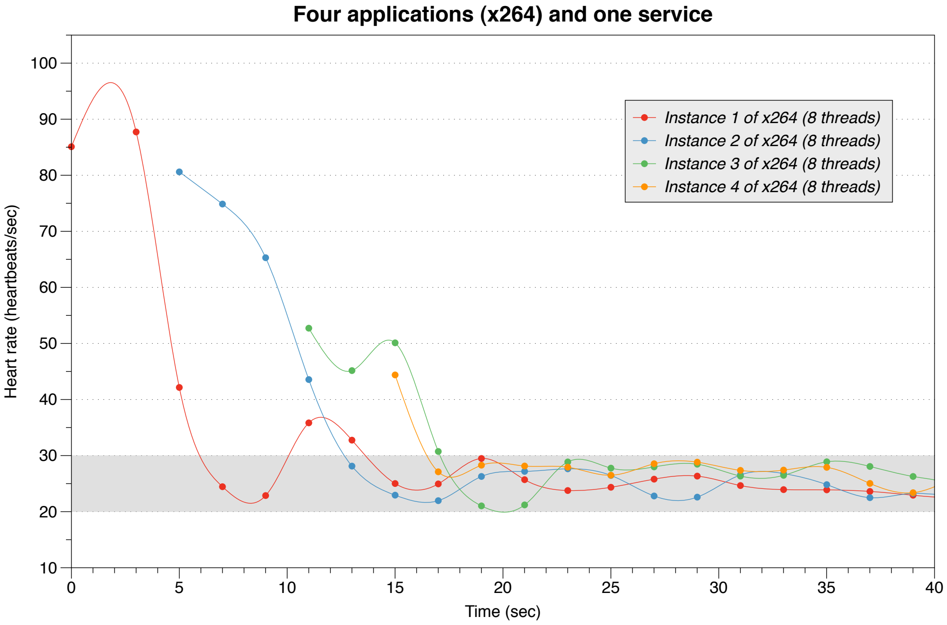


Figure 5.19: Four instances of `x264` started five seconds one after the other. They have all declared that they wish to produce 20 to 30 heartbeats per second (shown in gray).

Analyzing huge solution spaces: eight applications

This test shows that even when the solution space is huge the decision engine is still able to choose actions in order to make the heart rate of the applications converge.

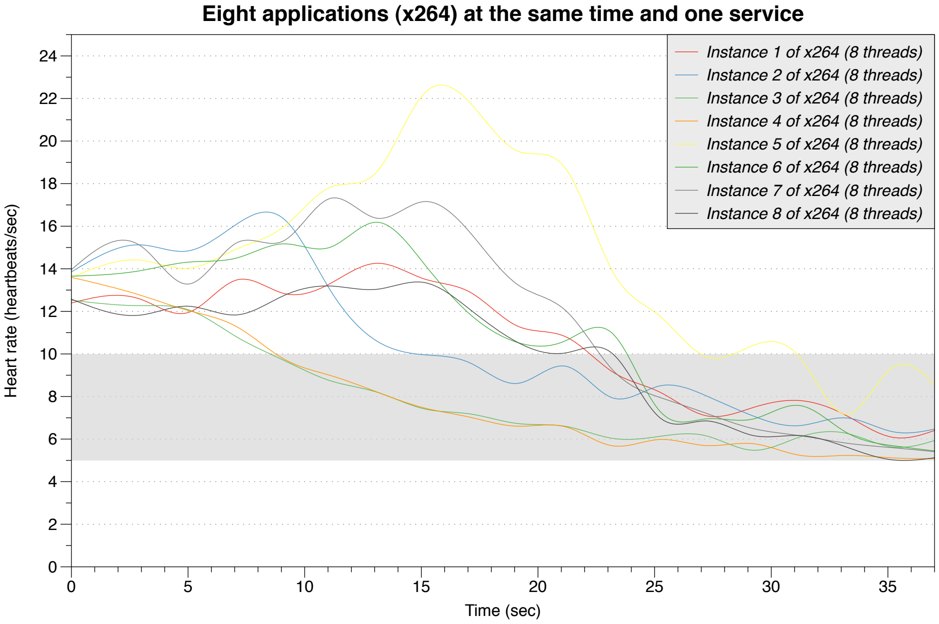


Figure 5.20: Eight instances of `x264` running at the same time and with an heart rate goal of 5 to 10 heartbeats per second (shown in gray).

Chapter 6

Conclusions and future works

"I came to the conclusion that
I am not a fiction writer."
Tim LaHaye

In this last Chapter the obtained results are summarized and future work is planned.

The aim of this thesis has been to design and implement a system-wide self-adaptive self-aware infrastructure. Today's systems complexity is sky-rocketing and there is a strong need for transferring into modern computing systems some of the knowledge that programmers have: the aim is to reduce the burden on them and automate several optimization without requiring manual intervention.

In this context we propose an autonomic architecture capable of monitoring its current status and its progress towards certain goals, capable of performing optimizations on itself, and capable of adapting to unpredictable, unknown, and unfavorable conditions through a decision engine that can learn from experience and optimize the performance of the system.

After stating the problem in Chapter 1 and extensively discussing the State of the Art in the field of autonomic computing in Chapter 2, we outline and throughly describe our solution in the remaining Chapters.

In Chapter 3 we present the idea behind our architecture: we aim at building a system that observes, decides, and act. We start from the *Applications Heartbeats* as a simple, flexible, and lightweight monitoring facility and we have literally built around it. We have developed two novel components: the *Consensus Object* and the *Services API*. The *Consensus Object* is the central entity that gathers all the information coming from Heartbeats-enabled applications and decides which actions to undertake in order to make the processes reach the desired goals; its decision engine is based on *R-learning*, a machine learning algorithm that makes it possible for the *Consensus Object* to learn from experience. The *Services API* is a programming interface suggested for those processes that can improve or decrease performance (either at system level and at application level): services adopting such interface are controlled by the *Consensus Object* which can enable them, disable them, or alter their behavior. The *Services API* is equal among all possible services and the *Consensus Object* uses the same commands to control each service. The decision engine will then understand which services to enable and when. In this way new services can be added extremely easily and the whole architecture proves to be modular, flexible, and extensible. The *Consensus Object* and the *Services API* implementations have been discussed in great detail in Chapter 4.

In Chapter 5 the entire architecture is put to test. Through a series of tests it has been proved that the whole infrastructure is lightweight and ready to be expanded in the future. In particular, the *Applications Heartbeats* overhead has been quantified as extremely small ($\sim$ 4% of the execution time). The overhead of the *Consensus Object* and of the *Services API* is also quite small: for a relatively simple service such as the *core allocator* the overhead is $\sim$ 15% of the execution time. Considering that most other services are either more complex (that is, they take longer to perform one iteration) or "sleep" for a few milliseconds between one iteration and the other, the average overhead will probably be even smaller. In fact the overhead on one iteration is constant and spending more time computing one iteration reduces the proportions of the overhead.

Moreover, we have built and tested an example of *Implementations Library* that explores some of the concepts developed in Chapter 3 and proves the usefulness of porting an autonomic infrastructure to reconfigurable hardware.

Finally the potential of the decision engine is tested: with either one or more Heartbeats-enabled programs running and either with the *core allocator* or an ad-hoc service, the *Consensus Object* is always capable of bringing the heart rate in the desired range.

Even if there is still a lot of work to do before the described architecture is mature, we have designed and built an *enabling technology* that is both novel and very promising. The developed APIs are complete and ready to be used. Future works include:

- the implementations of more services
- the extension of the *Applications Heartbeats* to support dynamic performance goals. This step is necessary to avoid situations like the one presented in Section 5.1.2; in such scenarios it is easy to imagine that the performance goals (and the issued heartbeats) divided by a factor in order to bring the *Applications Heartbeats* overhead at a minimum
- the porting of the framework to the Linux kernel, in order to have more control on the scheduling of threads and processes
- the development of a non-centralized scenario where the *Consensus Object* is distributed among the services instead of being central

Bibliography

[1] Anant Agarwal, Marco D. Santambrogio, David Wingate, and Jonathan Eastep. Smartlocks: Self-aware synchronization through lock acquisition scheduling. Technical report, MIT CSAIL, November 2009.

[2] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT CSAIL, September 2009.

[3] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application heartbeats for software performance and health. Technical report, MIT CSAIL, 2009.

[4] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application heartbeats for software performance and health. *15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 347–348, January 2010.

[5] Andrew Baumann University, Andrew Baumann, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, 2004.

[6] Ebeling William H. C. and Borriello Gaetano. Field programmable gate array. U.S. Patent 5208491, Washington Research Foundation, 1992.

[7] Marco D. Santambrogio. From reconfigurable architectures to self-adaptive autonomic systems. In *CSE '09: Proceedings of the 2009 International Conference on Computational Science and Engineering*, pages 926–931, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. In *PROCEEDINGS OF THE IEEE*, pages 615–638, 1998.

[9] Pao-Ann Hsiung, Marco D. Santambrogio, and Chun-Hsian Huang. *Reconfigurable System Design and Verification*. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[10] Vincenzo Rana, Marco D. Santambrogio, and Donatella Sciuto. Dynamic reconfigurability in embedded system design. In *ISCAS*, pages 2734–2737, 2007.

[11] K. Paulsson, M. Hübner, and J. Becker. On-line optimization of fpga power-dissipation by exploiting run-time adaption of communication primitives. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 173–178, New York, NY, USA, 2006. ACM.

[12] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. A unified operating system for clouds and manycore: fos. *1st Workshop on Computer Architecture and Operating System co-design (CAOS)*, January 2010.

[13] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX*

conference on Operating Systems Design and Implementation, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[14] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 35–44, New York, NY, USA, 2002. ACM.

[15] Trevor Pering, Tom Burd, and Robert Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *In Power Driven Microarchitecture Workshop, attached to ISCA98*, 1998.

[16] Autonomic computing: Ibm's perspective on the state of information technology.

[17] Winston Royce. Managing the development of large software systems. *Proceedings of IEEE WESCON*, pages 1–9, August 1970.

[18] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39:36–43, 2006.

[19] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[21] David Pescovitz. Monsters in a box, December 2000. `http://www.wired.com/wired/archive/8.12/supercomputers.html`.

[22] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[23] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

[24] Robert Laddaga. Active software. In *IWSAS*, pages 11–26, 2000.

[25] Robert Laddaga. Self-adaptive software. Technical report, DARPA Broad Agency Announcement (BAA), 1997.

[26] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[27] Ibm paves the way for mainstream adoption of autonomic computing.

[28] Dynamic systems initiative.

[29] Adaptive enterprise.

[30] Sun n1 grid engine 6.

[31] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. In *ICAS '08: Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, pages 175–181, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Jim Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, Department of Computer Science, University of Dublin, Trinity College, 2004.

[33] Michael G. Hinchey and Roy Sterritt. Self-managing software. *Computer*, 39(2):107, 2006.

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

[35] Hua Liu and Manish Parashar. Rule-based monitoring and steering of distributed scientific applications. *Int. J. High Perform. Comput. Netw.*, 3(4):272–282, 2005.

[36] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 172, Washington, DC, USA, 1998. IEEE Computer Society.

[37] D. M. Ogle, K. Schwan, and Richard Thomas Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Trans. Parallel Distrib. Syst.*, 4(7):762–778, 1993.

[38] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Trans. Comput. Syst.*, 6(2):157–195, 1988.

[39] Michael J. Kaelbling and David M. Ogle. Minimizing monitoring costs: Choosing between tracing and sampling. In *In Proceedings of the 23rd International Hawaii Conference on System Sciences*, volume I, pages 314–320, 1990.

[40] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[41] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[42] Andrew Baumann and Jeremy Kerr. Module hot-swapping for dynamic update and reconfiguration in k42.

[43] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING*, pages 1–27. IEEE Computer Society, 1998.

[44] Matteo Frigo and Steven G. Johnson. Fftw: An adaptive software architecture for the fft. pages 1381–1384. IEEE, 1998.

[45] Xiaoming Li, Maria Jesus Garzaran, and David Padua. A dynamically tuned sorting library, 2004.

[46] Johnathan Eastep, Harshad Kasture, and Anant Agarwal. The organic template library: A parallel runtime-adaptive C++ library.

[47] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288, New York, NY, USA, 2005. ACM.

[48] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation Due to copyright restrictions we are not able to make the PDFs for this conference available for downloading*, pages 89–104, New York, NY, USA, 2002. ACM.

[49] Craig A. N. Soules, Dilma Da Silva, Marc Auslander, Gregory R. Ganger, and Michal Ostrowski. System support for online reconfiguration. In *In Proc. USENIX Annual Technical Conference*, pages 141–154, 2003.

[50] An artificial intelligence perspective on autonomic computing policies. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, page 3, Washington, DC, USA, 2004. IEEE Computer Society.

[51] Hua Liu. A component-based programming model for autonomic applications. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 10–17, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. Qos aspect languages and their runtime integration. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 303–318, London, UK, 1998. Springer-Verlag.

[53] S. Agarwala, Yuan Chen, D. Milojicic, and K. Schwan. Qmon: Qos- and utility-aware monitoring in enterprise systems. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, pages 124–133, Washington, DC, USA, 2006. IEEE Computer Society.

[54] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Adv. Eng. Inform.*, 19(3):181–187, 2005.

[55] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical model-based autonomic control of software systems. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[56] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[57] Gabor Karsai, Ákos Lédeczi, Janos Sztipanovits, Gábor Péceli, Gyula Simon, and Tamás Kovácsházy. An approach to self-adaptive software based on supervisory control. In *IWSAS*, pages 24–38, 2001.

[58] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:2004, 2003.

[59] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. Monalisa : A distributed monitoring service architecture, 2003.

[60] David L. Oppenheimer, Vitaliy Vatkovskiy, Hakim Weatherspoon, Jason Lee, David A. Patterson, and John Kubiatowicz. Monitoring, analyzing, and controlling internet-scale systems with acme. *CoRR*, cs.DC/0408035, 2004.

[61] HP openview. `http://www.openview.hp.com`.

[62] IBM tivoli monitoring. `http://www-01.ibm.com/software/tivoli/products/monito`

[63] Jeffrey S. Vetter and Karsten Schwan. High performance computational steering of physical simulations. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, page 128, Washington, DC, USA, 1997. IEEE Computer Society.

[64] Weiming Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: on-line monitoring and steering of large-scale parallel programs. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Fron-*

tiers'95), page 422, Washington, DC, USA, 1995. IEEE Computer Society.

[65] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 9–20, Berlin, Heidelberg, 2009. Springer-Verlag.

[66] Daniel Reed, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha, Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty. Scalable performance environments for parallel systems. In *In Proceedings of the Sixth Distributed Memory Computing Conference (1991), IEEE Computer*, pages 562–569. Society Press, 1991.

[67] David R. Kohr, Jr., Xingbin Zhang, Daniel A. Reed, and Mustafizur Rahman. A performance study of an object-oriented, parallel operating system. In *In Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 27–2000. IEEE Computer Society Press, 1994.

[68] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in active harmony. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 180, Washington, DC, USA, 1998. IEEE Computer Society.

[69] W. E. Cohen. Multiple architecture characterization of the linux build process with oprofile. Technical report, IEEE, 2003.

[70] Jasmina Jancic, Christian Poellabauer, Karsten Schwan, Mathhew Wolf, and Neil Bright. dproc - extensible run-time resource monitoring for cluster applications. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 894–903, London, UK, 2002. Springer-Verlag.

[71] From Kernel To, Tom Zanussi, Karim Yaghmour, and Robert Wisniewski. relayfs: An Efficient Unified Approach for Transmitting Data. In *In Proceedings of the Ottawa Linux Symposium 2003*, pages 494–507, 2003.

[72] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[73] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[74] *An architectural blueprint for autonomic computing*, 2006.

[75] Robert W. Wisniewski, Peter F. Sweeney, Kartik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Calin Cascaval, and Reza Azimi. Performance and environment monitoring for whole-system characterization and optimization.

[76] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal, and Sun Microsystems. Dynamic instrumentation of production systems. pages 15–28, 2004.

[77] Sharon E. Perl and William E. Weihl. Performance assertion checking. *SIGOPS Oper. Syst. Rev.*, 27(5):134–145, 1993.

[78] Mark E. Crovella, Jeffrey K. Hollingsworth, Michael J. Kaelbling, David M. Ogle Min, Ten hwang Lai, Sartaj Sahni Anomalies, Ted Lehr, Zary Segall, Dalibor Vrsalovic, Margaret Martonosi, Anoop Gupta,

Barton P. Miller, Barton P. Miller, Morgan Clark, Jeff Hollingsworth, and Steven Kierstead. Performance debugging using parallel performance predicates, 1993.

[79] David Breitgand, Maayan Goldstein, Ealan Henis, Onn Shehory, and Yaron Weinsberg. Panacea towards a self-healing development framework. In *Integrated Network Management, IM 2007. 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany, 21-25 May 2007*, pages 169–178. IEEE, 2007.

[80] Jeffrey S. Vetter and Patrick H. Worley. Asserting performance expectations. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[81] Janak Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[82] Intel Pin. `http://www.pintool.org`.

[83] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *Int. J. High Perform. Comput. Appl.*, 13(3):263–276, 1999.

[84] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE COMPUTER*, 28:37–46, 1995.

[85] Bryan Buck Jeffrey and Jeffrey K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.

[86] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2007. ACM.

[87] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Automatic on-line failure diagnosis at the end-user site. In *In HotDep*, 2006.

[88] Michael Martin, V. Benjamin, Livshits Monica, and S. Lam. Finding application errors using pql: a program query language. In *In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 365–383. ACM Press, 2005.

[89] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *In Proc. OSDI*, 2006.

[90] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.

[91] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[92] Robert W. Wisniewski, Dilma da Silva, Marc Auslander, Orran Krieger, Michal Ostrowski, and Bryan Rosenburg. K42: lessons for the os community. *SIGOPS Oper. Syst. Rev.*, 42(1):5–12, 2008.

[93] O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski J. W., Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. pages 133–145, 2006.

[94] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.

[95] Anton Schwartz. A Reinforcement Learning Method for Maximizing Undiscounted Rewards. In *ICML*, pages 298–305, 1993.

[96] Data Encryption Standard (DES), U.S. Department of Commerce / National Institute of Standards and Technology, 1999.

Vita

Name	*Marco Triverio*
Birth date	December 10, 1986
Email	marco.triverio@gmail.com
Website	trive.110mb.com
Mobile	+39 338 9478800

Education

2010–2011 **Master in Interaction Design**, Copenhagen Institute of Interaction Design. *Ongoing*.

2009–2010 **Master of Science in Computer Science**, University of Illinois at Chicago, *GPA 3.86/4*.

Application heartbeats: a technique for enhancing system self-adaptability

2008–2010 Laurea Specialistica degree (equivalent to **Master of Science in Computer Engineering**), Politecnico di Milano, Italy, *GPA 28.91/30*.

Self-adaptability via heartbeats and central multi-application coordination

2008–present **Alta Scuola Politecnica** degree, Politecnico di Milano / Politecnico di Torino. Date of graduation: December 2010
D-sign: an hub for fashion talents

2005–2008 Laurea Triennale degree (equivalent to **Bachelor of Science in Computer Engineering**), Politecnico di Milano, Italy, *Summa Cum Laude*.
Design and implementation of the control system for an inverse pendulum

2001–2005 High School Diploma and Certificate of Excellence, Liceo Scientifico A. Avogadro, Biella, Italy, *100 out of 100*

2003 Attendance at Mairehau High School, Christchurch, New Zealand.
Historical and mathematical aspects of cryptography

Research and Publications

2010 *Playing seriously with mitigation strategies for climate change: the PolyGame*, Talk at the "Italian Ecology Society" conference, Collaborators: Renato Casagrnadi, Giulia Fiorese, Martina Gallia, Matteo Giuliani, Marko Radeta, Mario Sangiorgio, Emanuele Martelli, Stefano Consonni, Stefano Caserini.

2010 *Self-aware adaptation in FPGA-based systems*, FPL 20th Conference, Collaborators: Filippo Sironi, Marco Santambrogio, Hank Hoffman, Martina Maggio.

2010 *Reinventing the wheel: Arduino-based interactive automatic composer of low-fi electronic music*, Collaborator: Davide Totaro.
Winner of Milano Green Contest and on exhibition at Milan's FuoriSalone 2010

2009–present *Application heartbeats: a technique for enhancing system self-adaptability*, Advisors: John Lillis and Marco Santambrogio, Master of Science thesis.

2009 *Preliminary Design of an Embodied, Multi-tiered Exploration of Communication among Bees*, Advisors: Tom Moher.

2009 *Engineering of a Laboratory Simulation and Design software*, work done for Inpeco s.r.l. company.

2007–2008 *Design and implementation of the control system for an inverse pendulum (Progetto e implementazione del sistema di controllo per un pendolo inverso)*, Advisors: Sergio Bittanti and Fabio Previdi, Bahcelor of Science thesis.

Work Experiences

2009–2010 **Designing the user experience of the PolyGame**, Alta Scuola Politecnica, Milan, Italy.

Reference: Renato Casagrandi, casagran@elet.polimi.it

2008–2009 **Feasibility study and software engineering**, Inpeco s.r.l., Milan, Italy.

Reference: Marco Santambrogio, santa@csail.mit.edu, +39 335 6847022

2007–2009 **IT Consultant**, MBS s.a.s., Biella, Italy.

Reference: Bianchetto Enrico, e.bianchetto@mbscolle.191.it, +39 335 6349984

2008 **Collaborator** for the “Big Book of Apple Hacks”, published by O’Reilly.

Reference: Chris Seibold, cseibold@mac.com

2007 **Internship**, Domina s.r.l., Biella, Italy.

Reference: Perona Massimo, mperona@dobi.it, +39 335 7720492

2005–2007 **Miscelleanea**, Collaborations with Hakin9, Applicando, and Hacker Journal. Speaker at the conference "*Free Software Free Thinking*"

Skills

Coding C/C++, Java, Objective-C, Cocoa, Processing, OpenGL (freglut), PHP, AppleScript, Bash scripting

Engineering UML, Unit-testing, JML, JUnit

Electronics Arduino, PIC

IDEs Xcode, Interface Builder, NetBeans, Eclipse

Graphicss Photoshop, Illustrator, OmniGraffle, Aperture, Lightroom

Office Microsoft Office, OpenOffice, Apple iWork (Pages, Keynote, Numbers)

Databases Business intelligence, SQL

OSes Mac OS X, Linux, Windows

Soft sociable, detail oriented, team player, elected team leader in several university projects, used to multidisciplinary environments, good speaker (in English too)

Additional information

Scolarships **UIC Best Student**, **Accenture 2008/2009**

Languages English (TOEFL: 107/120, 2008 — CAE: Grade B, 2006), Italian (mother tongue)

Interests Music, photography, electronics

Sports Skiing, mountain-biking