Enhancing Self-Adaptive Computing Systems via Artificial Intelligence

Techniques and Active Learning

BY

JACOPO PANERATI
B.S. in Computer Engineering, Politecnico di Milano, 2009

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2012

Chicago, Illinois

Defense Committee:

Piotr Gmytrasiewicz, Chair and Advisor
John Lillis
Marco D. Santambrogio, Politecnico di Milano
To someone.
ACKNOWLEDGMENTS

I want to thank all the people who helped me with this in the last years and months. My advisor in Milan, Marco D. Santambrogio and professor Piotr in Chicago, their support just went so much beyond this thesis. All of the guys in the lab, Filippo, DBB, Carteo, il Catta, Martina, Fabio, Domo, Raffa, Focaccina and whoever else does not come to my mind now because I’m writing in a sleepy November morning. My pipz in Chicago, Albe, Fra, Anto, Ciube, my roommate Gary, for quietly supporting the Cowboys, the UIC Flames XC team, for every day we ran 16 miles, the Chicago master swimming team, because 5:20am swimming is true dedication and those guys at Caribou Coffee for supporting my caffeine addiction. Finally my former roommate Cisco and everyone else has been around in these years for any reason. Sometimes it might not look that way, but I am thankful.

JP
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><strong>INTRODUCTION</strong></td>
</tr>
<tr>
<td>1.1</td>
<td>Autonomic Computing</td>
</tr>
<tr>
<td>1.1.1</td>
<td>The Word “Autonomic”</td>
</tr>
<tr>
<td>1.1.2</td>
<td>The Need for Autonomic Computing</td>
</tr>
<tr>
<td>1.1.2.1</td>
<td>Moore’s Law and Software Systems</td>
</tr>
<tr>
<td>1.2</td>
<td>Goal of Autonomic Computing: Self-Management</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Autonomic Computing Terminology</td>
</tr>
<tr>
<td>1.3</td>
<td>Architectures of Autonomic Computing Systems</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Autonomic Element Control Loop</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Elements of a Typical Intelligent Agent</td>
</tr>
<tr>
<td>1.4</td>
<td>Purpose of this Work</td>
</tr>
<tr>
<td>1.5</td>
<td>Theoretical Challenges</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Machine Learning in Complex Systems</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td><strong>THEORETICAL BACKGROUND</strong></td>
</tr>
<tr>
<td>2.1</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Environments and Their Properties</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Types of Agent</td>
</tr>
<tr>
<td>2.2</td>
<td>Making Decisions</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Utility Theory</td>
</tr>
<tr>
<td>2.3</td>
<td>Complex Decision Making and Planning: Markov Decision Processes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Elements of an MDP</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Policies</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Complexity of MDPs</td>
</tr>
<tr>
<td>2.4</td>
<td>Active Reinforcement Learning</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Learning a Model and Utilities using Adaptive Dynamic Programming</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Model-Free Methods</td>
</tr>
<tr>
<td>2.4.2.1</td>
<td>Temporal Difference Methods in Passive Learning</td>
</tr>
<tr>
<td>2.4.2.2</td>
<td>Q-Learning</td>
</tr>
<tr>
<td>2.4.2.3</td>
<td>SARSA</td>
</tr>
<tr>
<td>2.5</td>
<td>Exploration</td>
</tr>
<tr>
<td>2.5.1</td>
<td>General Multi-State Problems: Exploration Vs. Exploitation</td>
</tr>
<tr>
<td>2.5.1.1</td>
<td>Undirected Techniques</td>
</tr>
<tr>
<td>2.5.1.2</td>
<td>Directed Techniques</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>3 RELATED WORK</td>
<td>48</td>
</tr>
<tr>
<td>3.1 Monitors: Making Systems Self- (and Context-) Aware</td>
<td>48</td>
</tr>
<tr>
<td>3.2 Decisions and Actions: Making Systems Self-Adaptive</td>
<td>52</td>
</tr>
<tr>
<td>3.3 The Previous AcOS Architecture</td>
<td>56</td>
</tr>
<tr>
<td>3.4 Artificial Intelligence in Autonomic Computing</td>
<td>60</td>
</tr>
<tr>
<td>3.4.1 Mixing Policies</td>
<td>62</td>
</tr>
<tr>
<td>3.5 Reinforcement Learning in Autonomic Computing</td>
<td>63</td>
</tr>
<tr>
<td>3.5.1 Challenges</td>
<td>64</td>
</tr>
<tr>
<td>3.6 Other Related Work</td>
<td>67</td>
</tr>
<tr>
<td>4 PROPOSED APPROACH</td>
<td>72</td>
</tr>
<tr>
<td>4.1 AcOS: Autonomic Operating System</td>
<td>73</td>
</tr>
<tr>
<td>4.1.1 How this Work Fits into AcOS</td>
<td>73</td>
</tr>
<tr>
<td>4.1.2 HRM</td>
<td>74</td>
</tr>
<tr>
<td>4.2 Proposed Architecture</td>
<td>75</td>
</tr>
<tr>
<td>4.2.1 Modularity</td>
<td>76</td>
</tr>
<tr>
<td>4.2.2 Acting</td>
<td>76</td>
</tr>
<tr>
<td>4.2.3 Sensing</td>
<td>77</td>
</tr>
<tr>
<td>4.2.4 Decision Making</td>
<td>78</td>
</tr>
<tr>
<td>4.3 Artificial Intelligence Approach</td>
<td>80</td>
</tr>
<tr>
<td>4.4 Novelty of the Proposed Approach</td>
<td>82</td>
</tr>
<tr>
<td>5 SYSTEM IMPLEMENTATION</td>
<td>84</td>
</tr>
<tr>
<td>5.1 Communication</td>
<td>84</td>
</tr>
<tr>
<td>5.1.1 AdaM-APs</td>
<td>85</td>
</tr>
<tr>
<td>5.1.2 AdaM/APs-HRM</td>
<td>90</td>
</tr>
<tr>
<td>5.2 Structure of a Generic AP</td>
<td>94</td>
</tr>
<tr>
<td>5.3 AdaM</td>
<td>98</td>
</tr>
<tr>
<td>5.4 Sensor: HRM</td>
<td>100</td>
</tr>
<tr>
<td>5.5 Actuator: Core Allocator with “Taskset” Command</td>
<td>101</td>
</tr>
<tr>
<td>5.6 Actuator: Frequency Scaler</td>
<td>104</td>
</tr>
<tr>
<td>5.7 Reinforcement Learning</td>
<td>106</td>
</tr>
<tr>
<td>6 TEST SET UP AND RESULTS</td>
<td>115</td>
</tr>
<tr>
<td>6.1 Testing Environment</td>
<td>115</td>
</tr>
<tr>
<td>6.2 PARSEC Benchmark Suite</td>
<td>116</td>
</tr>
<tr>
<td>6.3 Tests Description</td>
<td>117</td>
</tr>
<tr>
<td>6.4 Tests Commentary</td>
<td>119</td>
</tr>
<tr>
<td>6.4.1 Blackscholes</td>
<td>119</td>
</tr>
<tr>
<td>6.4.2 Bodytrack</td>
<td>121</td>
</tr>
<tr>
<td>6.4.3 Canneal</td>
<td>121</td>
</tr>
<tr>
<td>6.4.4 Dedup</td>
<td>124</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.5</td>
<td>Ferret</td>
</tr>
<tr>
<td>6.4.6</td>
<td>Fluidanimate</td>
</tr>
<tr>
<td>6.4.7</td>
<td>Raytrace</td>
</tr>
<tr>
<td>6.4.8</td>
<td>Swaptions</td>
</tr>
<tr>
<td>6.4.9</td>
<td>x264</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison with Previous Work</td>
</tr>
<tr>
<td>7</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
</tr>
<tr>
<td>7.1</td>
<td>Final Considerations</td>
</tr>
<tr>
<td>7.2</td>
<td>Research Development</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Further Testing</td>
</tr>
<tr>
<td>7.2.2</td>
<td>POMDPs</td>
</tr>
<tr>
<td></td>
<td>CITED LITERATURE</td>
</tr>
<tr>
<td></td>
<td>VITA</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>STATE OF THE ART APPROACHES COMPARISON</td>
<td>71</td>
</tr>
<tr>
<td>II</td>
<td>COMPARISON OF PREVIOUS APPROACHES WITH THIS WORK</td>
<td>83</td>
</tr>
<tr>
<td>III</td>
<td>COMPARISON OF TESTED AGENTS BY AVERAGE HEAR RATE (HB/S) AND ERROR (ADIMENSIONAL)</td>
<td>133</td>
</tr>
<tr>
<td>IV</td>
<td>COMPARISON OF AGENTS, CONTROL THEORY AND HEURISTICS BY AVERAGE HEAR RATE (HB/S) AND ERROR (HB/S)</td>
<td>137</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Moore’s law: original 1965 graph, artwork celebrating its 40th anniversary and the decrease of transistors cost, from <a href="http://www.intel.com">www.intel.com</a></td>
</tr>
<tr>
<td>2</td>
<td>Structure of an autonomic element in the context of an autonomic system, from “The vision of autonomic computing” (2003)</td>
</tr>
<tr>
<td>3</td>
<td>The MAPE loop in an autonomic element/agent, from “Reinforcement Learning in Autonomic Computing” (2007)</td>
</tr>
<tr>
<td>5</td>
<td>A generic agent, from “Artificial Intelligence: a Modern Approach”</td>
</tr>
<tr>
<td>6</td>
<td>Representation of the use of the heartbeats for self-awareness or external monitoring, from “Application Heartbeats” (2010)</td>
</tr>
<tr>
<td>7</td>
<td>Representation of the original architecture, from “Application heartbeats: a technique for enhancing system self-adaptability” (2010)</td>
</tr>
<tr>
<td>8</td>
<td>Hierarchy new architecture</td>
</tr>
<tr>
<td>9</td>
<td>Communication flows</td>
</tr>
<tr>
<td>11</td>
<td>Blackscholes, ADP agent, first set of actions, avg. hr 5490257, % error 0.23</td>
</tr>
<tr>
<td>12</td>
<td>Blackscholes, ADP agent, second set of actions, avg. hr 10698401, % error 0.16</td>
</tr>
<tr>
<td>13</td>
<td>Blackscholes, ADP agent, third set of actions, avg. hr 9648170, % error 0.11</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>14</td>
<td>Blackscholes, Q-learning agent, first set of actions, avg. hr 9782523, % error 0.14</td>
</tr>
<tr>
<td>15</td>
<td>Blackscholes, Q-learning agent, second set of actions, avg. hr 10614396, %err 0.12</td>
</tr>
<tr>
<td>16</td>
<td>Blackscholes, Q-learning agent, third set of actions, avg. hr 7960482, % error 0.12</td>
</tr>
<tr>
<td>17</td>
<td>Bodytrack, ADP agent, first set of actions, avg. hr 1.7, % error 0.21</td>
</tr>
<tr>
<td>18</td>
<td>Bodytrack, ADP agent, second set of actions, avg. hr 1.8, % error 0.18</td>
</tr>
<tr>
<td>19</td>
<td>Bodytrack, ADP agent, third set of actions, avg. hr 1.2, % error 0.22</td>
</tr>
<tr>
<td>20</td>
<td>Bodytrack, Q-learning agent, first set of actions, avg. hr 1.8, % error 0.18</td>
</tr>
<tr>
<td>21</td>
<td>Bodytrack, Q-learning agent, second set of actions, avg. hr 2.2, % error 0.19</td>
</tr>
<tr>
<td>22</td>
<td>Bodytrack, Q-learning agent, third set of actions, avg. hr 1.8, % error 0.20</td>
</tr>
<tr>
<td>23</td>
<td>Canneal, ADP agent, first set of actions, avg. hr 979257, % error 0.11</td>
</tr>
<tr>
<td>24</td>
<td>Canneal, ADP agent, second set of actions, avg. hr 1125487, % error 0.11</td>
</tr>
<tr>
<td>25</td>
<td>Canneal, ADP agent, third set of actions, avg. hr 1075179, % error 0.11</td>
</tr>
<tr>
<td>26</td>
<td>Canneal, Q-learning agent, first set of actions, avg. hr 1121591, % error 0.12</td>
</tr>
<tr>
<td>27</td>
<td>Canneal, Q-learning agent, second set of actions, avg. hr 1130942, % error 0.12</td>
</tr>
<tr>
<td>28</td>
<td>Canneal, Q-learning agent, third set of actions, avg. hr 863815, % error 0.10</td>
</tr>
<tr>
<td>29</td>
<td>Dedup, ADP agent, first set of actions, avg. hr 5982, % error 0.49</td>
</tr>
<tr>
<td>30</td>
<td>Dedup, ADP agent, second set of actions, avg. hr 6386, % error 0.57</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>31</td>
<td>125</td>
</tr>
<tr>
<td>32</td>
<td>125</td>
</tr>
<tr>
<td>33</td>
<td>125</td>
</tr>
<tr>
<td>34</td>
<td>125</td>
</tr>
<tr>
<td>35</td>
<td>126</td>
</tr>
<tr>
<td>36</td>
<td>126</td>
</tr>
<tr>
<td>37</td>
<td>126</td>
</tr>
<tr>
<td>38</td>
<td>126</td>
</tr>
<tr>
<td>39</td>
<td>126</td>
</tr>
<tr>
<td>40</td>
<td>126</td>
</tr>
<tr>
<td>41</td>
<td>128</td>
</tr>
<tr>
<td>42</td>
<td>128</td>
</tr>
<tr>
<td>43</td>
<td>128</td>
</tr>
<tr>
<td>44</td>
<td>128</td>
</tr>
<tr>
<td>45</td>
<td>128</td>
</tr>
<tr>
<td>46</td>
<td>128</td>
</tr>
<tr>
<td>47</td>
<td>129</td>
</tr>
<tr>
<td>48</td>
<td>129</td>
</tr>
<tr>
<td>49</td>
<td>129</td>
</tr>
<tr>
<td>FIGURE</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>50</td>
<td>Raytrace, Q-learning agent, first set of actions, avg. hr 6.1, % error 0.19</td>
</tr>
<tr>
<td>51</td>
<td>Raytrace, Q-learning agent, second set of actions, avg. hr 7.0, % error 0.14</td>
</tr>
<tr>
<td>52</td>
<td>Raytrace, Q-learning agent, third set of actions, avg. hr 5.8, % error 0.19</td>
</tr>
<tr>
<td>53</td>
<td>Swaptions, ADP agent, first set of actions, avg. hr 39928, % error 0.09</td>
</tr>
<tr>
<td>54</td>
<td>Swaptions, ADP agent, second set of actions, avg. hr 48533, % error 0.10</td>
</tr>
<tr>
<td>55</td>
<td>Swaptions, ADP agent, third set of actions, avg. hr 47077, % error 0.10</td>
</tr>
<tr>
<td>56</td>
<td>Swaptions, Q-learning agent, first set of actions, avg. hr 30066, % error 0.20</td>
</tr>
<tr>
<td>57</td>
<td>Swaptions, Q-learning agent, second set of actions, avg. hr 49129, % error 0.11</td>
</tr>
<tr>
<td>58</td>
<td>Swaptions, Q-learning agent, third set of actions, avg. hr 44340, % error 0.11</td>
</tr>
<tr>
<td>59</td>
<td>x264, ADP agent, first set of actions, avg. hr 7.35, % error 0.29</td>
</tr>
<tr>
<td>60</td>
<td>x264, ADP agent, second set of actions, avg. hr 8.26, % error 0.27</td>
</tr>
<tr>
<td>61</td>
<td>x264, ADP agent, third set of actions, avg. hr 7.48, % error 0.28</td>
</tr>
<tr>
<td>62</td>
<td>x264, Q-learning agent, first set of actions, avg. hr 8.76, % error 0.42</td>
</tr>
<tr>
<td>63</td>
<td>x264, Q-learning agent, second set of actions, avg. hr 9.15, % error 0.47</td>
</tr>
<tr>
<td>64</td>
<td>x264, Q-learning agent, third set of actions, avg. hr 7.95, % error 0.39</td>
</tr>
<tr>
<td>65</td>
<td>Bodytrack, comparison of a trained agent with control theory and heuristic approaches</td>
</tr>
<tr>
<td>66</td>
<td>Fluidanimate, comparison of a trained agent with control theory and heuristic approaches</td>
</tr>
<tr>
<td>67</td>
<td>Raytrace, comparison of a trained agent with control theory and heuristic approaches</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>AC</td>
<td>Autonomic Computing</td>
</tr>
<tr>
<td>AcOS</td>
<td>Autonomic Operating System</td>
</tr>
<tr>
<td>AdaM</td>
<td>Adaptation Manager</td>
</tr>
<tr>
<td>ADP</td>
<td>Adaptive Dynamic Programming</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AP</td>
<td>Adaptation Policy</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CA</td>
<td>Core Allocator</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FS</td>
<td>Frequency Scaler</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GID</td>
<td>Group Identifier</td>
</tr>
<tr>
<td>HB</td>
<td>Heartbeat</td>
</tr>
<tr>
<td>HRM</td>
<td>Hear-Rate Monitor</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>MAPE</td>
<td>Monitor-Analyze-Plan-Execute</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Process</td>
</tr>
<tr>
<td>ODA</td>
<td>Observe-Decide-Act</td>
</tr>
<tr>
<td>PID</td>
<td>Process Identifier</td>
</tr>
<tr>
<td>POLIMI</td>
<td>Politecnico di Milano</td>
</tr>
<tr>
<td>POMDP</td>
<td>Partially Observable Markov Decision Process</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>SARSA</td>
<td>State-Action-Reward-State-Action</td>
</tr>
<tr>
<td>TD</td>
<td>Temporal Difference</td>
</tr>
<tr>
<td>TID</td>
<td>Thread Identifier</td>
</tr>
<tr>
<td>UIC</td>
<td>University of Illinois at Chicago</td>
</tr>
<tr>
<td>uLab</td>
<td>Micro-Architectures Laboratory</td>
</tr>
</tbody>
</table>
SUMMARY

Autonomic computing (AC) has been proposed as a solution to the increasing complexity of computer systems, threatening to make systems impossible to be managed by human operators in the near future (1).

This work is an attempt to implement autonomicity to a certain extent (self-configuration, self-optimization, self-awareness) in a desktop computer operating system.

The main focus is on the monitors/sensors to be used in order to make the system self-aware, and the policies to use in order to make the system self-optimizing.

Two are the main goals of this work: 1) provide an updated, extended and unifying framework with respect to works such as (2), (3) and (4), here the monitoring is provided by the application programming interface (API) of heart rate monitor (HRM) (5) (an improved version of application heartbeat (3)) and code conventions are given to easily introduce new decision policies (combining what is good of (2) and (4)), moreover, 2) the new framework is made even more versatile by introducing the possibility to learn new policies on-line drawing ideas from artificial intelligence and reinforcement learning (translating to operating systems concepts preliminary explored in data centers by (6) and (7)).

We provide a detailed description of the implemented architecture and experimental results of running benchmarks from the PARSEC suite (8) on a multi-core system that implements the HRM monitor and uses policies derived from AI and learning to determine resource allocation.

This remainder of the dissertation is organized as follows.
SUMMARY (Continued)

Chapter 1 provides a general introduction on autonomic computing and its history, the reasons behind its introduction and the basic terminology needed to understand the rest of the work.

Chapter 2 contains the background knowledge derived from the field of artificial intelligence (AI) needed to understand this work, from the basic definitions.

Chapter 3 describes the state of art in the field of autonomic computing, especially through works that are strongly related to this one. We pass through the introduction of application heartbeat (3), and some of its applications (9), (4), (10). about environments and agents to the theory behind Markov decision processes, reinforcement learning algorithms and exploration.

Chapter 4 contains the proposed approach of this work, its inspiration, its architecture, what it draws from the previous one, in what it differs and why it should be better.

Chapter 5 is dedicated to the implementation details of the proposed approach, functions documentation and pseudo-code examples are provided here.

Chapter 6 describes the setting up of the tests for the given implementation, the hardware we used to perform these tests, their numerical results and graphics.

Chapter 7 contains our conclusions and thoughts about the results collected and our ideas about what further experiments we would like to explore and what should be developed next.
CHAPTER 1

INTRODUCTION

“Awake. Where am I?”

Memento (2000)

The research work introduced in these pages has its place to the broader world of autonomic computing systems, specifically, with the concepts of self-awareness, self-adaptation and self-management that have often been associated to this kind of systems (1).

Furthermore, it is worth notice how autonomic computing systems architectures have elements of resemblance (“Each element must include sensors and effectors” (11), (12)) with typical artificial intelligent agents (13).

This work proposes the implementation of complex (including the abilities of planning and learning (13)) intelligent agents in order to make “autonomic” (w.r.t. the properties described in (1) and (14)) a Unix based operating system.

The first section of this chapter is dedicated to autonomic computing in general and its history, then we discuss about the terminology used in this field of computer science and about the architectures proposed to implement autonomic computing systems. Finally, we present the purpose of this work and the several engineering and theoretical challenges brought by autonomic computing systems implementation.
1.1 Autonomic Computing

Autonomic computing is a fairly recent concept in computer science. Basic concepts were introduced in (15) and its fundamentals ideas were later developed in (1) and (12).

Autonomic computing is a “new approach to the design of computing systems” (14), and its goals/objectives includes, among others, the ability, for the system, to be “selfconfiguring, [...] and selfoptimizing” (14), self-adaptive as well as “self-protecting and self-healing” (11).

From 2001, IBM publications about autonomic computing (in particular the one from Watson laboratory) provide some general elements of the desired architectures for autonomic systems and the main challenges of realizing the autonomic evolution (and not revolution (12)) are developed.

1.1.1 The Word “Autonomic”

Autonomic (adj) acting or occurring involuntarily (autonomic reflexes), first known use 1898 (from Merriam Webster dictionary).

The term “Autonomic” was used by IBMs senior vice president of research, Paul Horn, in March 2001 while introducing this concept at the National Academy of Engineers at Harvard University (1). This term has a strong biological connotation, “autonomic nervous system governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions” (1).

By using the term “Autonomic”, IBM wanted to suggest that looking at socio/biological systems of any scale was the right way to seek inspiration in the designing of future computer systems.
1.1.2 The Need for Autonomic Computing

Growing complexity is the main reason why IBM pointed out the necessity for a new way of designing computing systems.

In (16), (17) from October 2001, IBM exposes its concern and preoccupation about the state of information technology industry. This manifesto warns about a possible crisis arising from the increasing software complexity.

IBM’s manifesto explain how and why, computing systems are evolving in a direction that make more and more difficult, for IT professionals, to “install, configure, tune, and maintain” (1) them.

“Computing systems complexity appears to be approaching the limits of human capability [...] As systems become more interconnected and diverse, architects are less able to anticipate and design interactions [...] ” (1)

This problem was not something new in the software engineering world, as Frederick P. Brooks, Jr. once observed: “Complexity is the business we are in, and complexity is what limits us.” (18)

In 2001, Paul Horn, while introducing the term “autonomic’ wanted to address a possible solution, a way out of the growing complexity problem.

“The information technology industry loves to prove the impossible possible. [...] The obstacle is complexity... Dealing with it is the single most important challenge facing the I/T industry.” (12)
Some clarifying examples of how expensive, inefficient and human error-prone the standard systems have become is reported in IBM’s (12):

- Nick Tabellion, CTO of Fujitsu Softek, said that for one dollar spent purchasing storage, you spend 9 to manage it;

- David J. Clancy, chief of the Computational Sciences Division at the NASA Ames Research Center, said that 40% of the groups software work is devoted to test but also that the test problem grows exponentially with the range of behavior of a system.

1.1.2.1 Moore’s Law and Software Systems

In (12), we can find an interesting consideration on the relation in between Moore’s Law and the necessity for new ways to design computing system, such as the autonomic approach.

The Moore’s Law, maybe the most famous law in the computer engineering field, states that processing power on a fixed area chip will double every 18 to 24 months. This observation was made in 1965 by Gordon E. Moore, then, the director of research and development at Fairchild Semiconductor Corporation and, later, co-founder of Intel Corporation.

While it does not seem likely to see any change in this growing processing power trend in the near future, if we won’t change the way in which we develop software systems, it will become harder and harder to fully and properly exploit this new resources. Furthermore, without more skilled professional, these systems would become soon impossible to configure and maintain.
1.2 Goal of Autonomic Computing: Self-Management

The main aspect of every autonomic computing system (11) is the fact that it is able of self-management.

An autonomic computing system, in a way similar to its biological counterpart, is independent and able to make decisions for itself in a large number of different situations.

Autonomic systems can run indifferently on many hardware architectures, no matter which are the specific resources provided by them and without the need of human intervention.

Independently from the conditions inside or outside the system, with changing workloads, settings and devices, autonomic systems can install themselves and also tune themselves to obtain improved performance.
Moreover, autonomic systems should be able to run in challenging situations. When an error occurs, the autonomic system is able to detect it, go back in the computation and revert the error.

When malicious softwares are threatening an autonomic system, it is able to identify and respond to the menace.

So, the ultimate goal, for this kind of systems, is becoming able to self-manage themselves, relegating human intervention only to very high-level directives.

1.2.1 Autonomic Computing Terminology

The goal of self-management in autonomic computing systems is obtained through the realization of different objectives (11). These are usually identified in: self-configuration, self-optimization, self-healing and self-protection (1).

**Self-configuration** One of the more challenging, error-prone and time-consuming issues with very complex systems its their installation. If we look at large corporate data centers, they usually are composed by many heterogeneous components such as routers, servers, etcetera from different vendors.

An autonomic computing system have the ability to configure itself, leaving to the user only the task to specify high-level preferences. The autonomic computing system should also be able to detect and incorporate in real time, without external intervention new devices, elements and capabilities.

**Self-optimization** With growing complexity, performance of software systems depends on more and more parameters. These parameters may allow the system to run properly and
Figure 2. Structure of an autonomic element in the context of an autonomic system, from “The vision of autonomic computing” (2003)
efficiently in a broad number of situations and condition but, on the other hand, it is now very hard if not impossible to dispose of the human expertise needed to tune all of them properly.

The autonomic computing system does not need the operator intervention to tune and optimize itself. The autonomic computing system has the knowledge and the ability to process it needed to perform this task autonomously and/or it has the ability to learn about itself and how to act. In order to do this, the autonomic system is self-monitoring and able to self-adapt.

**Self-healing** In order to be considered truly autonomic, a modern computing system should be able to detect, identify and fix problems that may appear in the system during the computation.

This is maybe one of the most challenging aspects of making a computing system “autonomic”. In (1), it is suggested that this property could be achieved by a software able to match a log of the occurred failure to all the available patches for the system, and install the right one.

**Self-protection** Finally, the autonomic computing system must be able to protect itself from external malicious attacks. Nowadays, many tools to protect computing systems exist (e.g. firewalls and anti-virus packages) and most of them are partially or mostly automated.

What is new, in the way an autonomic computing system protect itself, is the fact that it protects the system as a whole and from a broad variety of attackers or problems.
Furthermore, the autonomic computing system, through its self-monitoring property, should be able to anticipate and avoid or mitigate (1) attacks.

It is important, dealing autonomic computing systems, to consider also dependability because “many areas of computing are addressing similar issues without being fully aware of related work in other fields and thus missing potential insights from that work” (14).

Dependability is a complex property of computing systems. It includes many attributes, e.g., reliability, availability, safety, security, survivability and maintainability (19).

In (20), Randell speaks about software faults and the misleading approach in which, computer scientists usually try to create error-less software instead of software able to cope with errors.

Even if some valuable communities exist in the field of fault-tolerant computing, (14) suggests that “with such systems becoming ever-more complex there is a growing need for developers to pay greater attention to dependability”.

The goal of self-management and the objectives described above can be achieved only if the systems has some specific properties that in (11) are described as:

**Self-awareness** the system is aware of its internal state.

**Environment-awareness** the system is aware of the changes happening around it.

**Self-monitoring** is the ability to detect the changes happening in the system and in the environment.

**Self-adjusting** (or self-adapting) is the ability to act accordingly with these changes.
This means that the system must have a relevant knowledge base, changing over time, and the ability to exploit it. Furthermore, in an heterogeneous/multi-agent environment, the system must be able to understand and communicate with other systems.

1.3 **Architectures of Autonomic Computing Systems**

Autonomic systems are intended as collections of autonomic elements (1). Each autonomic element has the ability to manage itself in a consistent way, making its decisions through some decision model (monitor-analyze-plan-execute, observe-decide-act, etcetera) and basing them on the knowledge of its internal status, the environment as well as optional built-in knowledge base.

But the property of the system of being autonomic does not arise just from the fact of each element being autonomic. In (1), it is stated that the self-management of the whole system will derive mainly from the large number of interactions in between these many autonomic elements.

A distributed and service-oriented infrastructure will be needed to support interaction of these elements.

Each autonomic element is identified and associated with its own autonomic manager.

1.3.1 **Autonomic Element Control Loop**

The inner workings of autonomic systems and elements are summarized in (12). “It is assumed that an autonomic computing system is made up of a connected set of autonomic elements. Each element must include sensors and effectors.” (12)

The sensors allows the autonomic element to be self-aware and monitor itself. Then, the autonomic element compares the data retrieved from the sensors with its expectations and
Figure 3. The MAPE loop in an autonomic element/agent, from “Reinforcement Learning in Autonomic Computing” (2007)

desires. If necessary, the autonomic element can decide which action to select and perform it through its effectors (or actuators), making it self-adapting. It is implied, as in the original IBM design for autonomic systems, that each autonomic element has an autonomic manager in order to be able to take decisions about the right action to choose.

Notice that sensors not only monitor the autonomic element itself, but also the environment in which the autonomic element works in.

From a logical point of view, most of autonomic elements implement a traditional “observe-decide-act” control loop (11).

1.3.2 Elements of a Typical Intelligent Agent

In this work, the autonomic computing system is realized and implemented through intelligent agents, as they are described in the field of artificial intelligence. For this reason, it is
An agent is “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators” (13).

An agent, similarly to a human being has:

- Sensors (a human has eyes and ears);
- Actuators (a human has hands and voice);

The agent bases its actions on the history of the perceptions it received.

“Mathematically speaking, we say that an agent’s behavior is agent function described by the agent function that maps any given percept sequence to an action.” (13)

We can easily see similarities with the monitors and actuators described in (12), with the concepts of self-awareness and self-adapting and the observe-decide-act control loop (11).
1.4 Purpose of this Work

This work is an attempt to implement autonomicity to a certain extent, through the aspects of self-configuration, self-optimization, self-awareness.

This objective is translated in the development of a software system in a Unix-based desktop-computer operating system, willing to create an autonomic element out of the operating system user-space.

The main focus is on the monitors/sensors to be used in order to make this system self-aware, and the policies to use in order to make the system self-optimizing.

Two are the main goals of this work:

1. Provide an updated, extended and unifying framework with respect to works such as (2), (3) and (4). Here the monitoring is provided by the application programming interface (API) of heart rate monitor (HRM) (5) (an improved version of application heartbeat (3)) and code conventions are given to easily introduce new decision policies (combining what is good of (2) and (4)).

2. The new framework is made even more versatile by introducing the possibility to learn new policies on-line drawing ideas from artificial intelligence and reinforcement learning (translating to operating systems concepts preliminary explored in data centers by (6) and (7)).

Experimental results are provided by running benchmarks from the PARSEC suite (8) on a multi-core system that implements the HRM monitor and uses policies derived from AI and learning to determine resource allocation.
1.5 **Theoretical Challenges**

The implementation of autonomic computing systems brings serious conceptual difficulties.

“We need fundamental mathematical work aimed at understanding how the properties of self-configuration, self-optimization, self-maintenance, and robustness arise from or depend on the behaviors, goals, and adaptivity of individual autonomic elements; the pattern and type of interactions among them; and the external influences or demands on the system. [...] The nonlinearity of emergent behavior makes such an inversion highly nontrivial.” (1)

Hints of possible solutions for these problems may come from different fields of science: control theory, artificial intelligence and machine learning, statistical modeling, negotiation theory.

In a recent article published on “Science” magazine, Jeffrey Kephart says: “few researchers have seriously attempted to develop computer analogs to the mechanisms employed by the autonomic nervous system, and only a handful have pursued approaches to self-managing computing that are in any way biological” (21). He also remarks the attempts done by researchers in the field of artificial neural networks (ANN).

1.5.1 **Machine Learning in Complex Systems**

Machine learning in single agent problems and relatively simple/stationary (not changing over time) environment is a quite well developed field.

It is worth notice that the environment in which autonomic elements are supposed to work is way more complex and challenging.
In fact, the typical environment for an autonomic computing element it is clearly populate by a large number of other agents (multiple agent problem) and it is inherently changing over time.

In such a difficult environment, in general, it is not possible to obtain some mathematical guarantees of learning algorithms. Still, learning techniques proved their effectiveness in many unsuspected situations (22).
CHAPTER 2

THEORETICAL BACKGROUND

“If such is the form of ultimate wisdom, then life is a greater riddle than some of us think it

to be.”

Heart of Darkness, Joseph Conrad (1902)

This chapter is dedicated to the background knowledge that is needed to better and fully
understanding the research work of this text. Many of the theoretical issues we will discuss are
derived from the field of artificial intelligence and we focus especially on decision making and
learning, two of the most complex and still unexplored aspects of this discipline.

We will start introducing artificial intelligence from the concepts of rational agents and the way
to characterize the environments they work in. Then we will move to the theories underlying
complex decision making that may require even the use of planning. We will discuss about
Markov property, Markov environments and Markov Decision Processes (MDPs). We will go
through the theorems and the algorithms that allow to solve this kind problem.

Finally, we will discuss the issue of learning in the context of Markov Decision Processes (MDPs)
and the challenging problem of exploration when learning has to be done on-line.
2.1 Artificial Intelligence

Artificial intelligence is a relatively young and developing field of science. According to (13), it starts to be a topic of discussion in the 40s drawing from neural physiology and Turing’s computing theories. Birth of artificial intelligence is established in 1956, when John McCarthy (Princeton, Stanford, Dartmouth College), together with Minsky, Claude Shannon, and Nathaniel Rochester organized a two-months workshop dedicated to this topic.

The definition of artificial intelligence, as well as the definition of mere intelligence, is not an easy task. Alan Turing attempted to give an operational definition of it through what is known as Turing test (1950).

“A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.” (13)

Still, a unique definition of what it means, to be artificially intelligent, is not known. Current definitions deals with the processes of thinking and acting and with the qualities of being human or rational (13).

However, the approach that is suggested in (13) and that we follow is the one that considers artificial intelligent machines as “rational agents”.

An agent is defined as any entity that is placed in an “environment”, can collects information about the environment through “sensors” and can operate on the environment through “actuators”.

An agent could have knowledge of the environment through what sensors are sensing at that moment, as well as what they sensed during the entire life span of the agent.
Agent’s behaviour is described/defined by a - so called - agent function. Conceptually, an agent function is nothing more than a table mapping states to actions. The “state” is defined as a function of the history of all that has been sensed by the agent during its history.

An intelligent agent is expected to be rational. Rationality of an agent (in choosing which action perform next) depends on different factors, the available actions in a given moment, what the agent knows of the environment from its sensor of what it knew a priori. In (13), however, it is stated that to define rationality we can not avoid to define/know a “performance measure”: a function to evaluate any sequence of states.

2.1.1 Environments and Their Properties

In (13) this description of agents in environments goes under the acronym of PEAS (Performance, Environment, Actuators, Sensors) description. Environments can be of very different nature, again in (13) a few dimension to characterize their main properties are suggested.

**Single Agent vs. Multi-Agent**: The first distinction we can made is between environment in which only one agent is present and environments in which multiple agents are performing actions. **Competitive vs. Cooperative**: Multi-Agent environments themselves can be sub-divided in environments in which the agents are pursuing a common goal or they are acting one against the other. **Uncertain Environments**: Some environments can contain some degree of uncertainty, this may come in two aspects. **Fully Observable vs. Partially Observable**: Some environments are fully observable in the sense that the sensors of an agent give it a complete picture of the environment. Sometimes, however, sensors are noisy or just missing and the environment is only partially observable by the agent. **Determin-
**Deterministic vs. Stochastic:** An environment is defined as deterministic if the next state of the system is univocally associated to the current state and the action performed. **Stationary vs. Non-Stationary:** As we just stated, the next state could be defined by a function (if the environment is deterministic) or a probability distribution (if the environment is stochastic). In any case, if this relation is not changing over time, the environment is called stationary. **Episodic vs. Sequential:** In sequential environments, any decision could affect all future decisions. In episodic environment, “agent’s experience is divided into atomic episodes” (13). Episodic environments, such as classification task, are typically much simpler. **Static vs. Dynamic:** A dynamic environment is an environment that changes even when the agent is not performing any action but it is just thinking about which action to take next. Dynamic environments are usually more complex to deal with. **Discrete vs. Continuous:** The property of being discrete rather than continuous may apply to several elements of the PEAS (Performance, Environment, Actuators, Sensors) description. Time, states and percepts could be discrete or continuous, but also actions could be continuous.

### 2.1.2 Types of Agent

As stated before, an agent is any entity that can collects information about the environment through “sensors” and can operate on the environment through “actuators”.

While sensors and actuators refer to the “architecture” of the agent, the behaviour of the agent is influenced only by its “agent function”.
In (13), a classification of agent functions suggest to distinguish four types of agents: simple reflex agents, model-based reflex agents, goal-based agents and utility-based agents. For the extent of this work, only the last two, the most complex, and the learning issue are relevant.

**Goal-Based Agents** Sometimes choosing which action to take next is not possible, intuitive or useful by only looking at the internal state. Sometimes, the agent only know about which state (or states) is desirable. This state is called goal or goal-state and these agents are called goal-based agents.

Goal-based agents require a quite relevant overhead of computation in deciding the action they will perform. Usually the exploit complex and computationally expensive algorithms to perform tasks such as “searching” and “planning”.

Even if less efficient, goal-based agents are more flexible and able to solve a broader and more complex population of problems.

**Utility-Based Agents** Utility-based agents are a generalization of goal-based agents. Utility-based agents are needed to overcome one major limitation of goal-based agents that could make them ineffective in some environments.

Goal-based agents make a rough separation between what is desirable (the goal states) and what is not, this simple binary classification could lead to the impossibility of making the right decision in some non-trivial situations.
Utility-based agents use a utility function (something similar to a built-in knowledge of the “performance measure” presented before) to make the correct decisions, trading off between more and less desirable situations.

In general, these complex agents move in complex environments that have elements of uncertainty and stochasticity. They will use complex probabilistic reasoning to choose their next action in order to maximize the expected utility they desire.

**Learning and Learning Agents** The types of agents presented by now requires different amount of knowledge base to be built-in in their agent function. Alan Turing, in 1950, considers this hypothesis, to build intelligent agents “by hand”, and he suggests that a “more expeditious method seems desirable” (13).

He suggests agents that are are able to learn. Learning agents are an extension of the agents we have discussed so far.

Learning agents are in general much more complex than the other rational agents we have summarized before and they are often presented with architectures that do not strictly follow the separation just exposed.

It is important to distinguish between agents that learn “on-line”, while performing their task, and agent that learn before being deployed in the actual environment.

### 2.2 Making Decisions

Intelligent, or rational, agents make decisions. These decisions usually concern the actions the agent will be willing to perform.
In general, these decisions are made in environments that might be contain aspects of uncertainty. This uncertainty, we said, comes in two forms. We have uncertainty due to partial or noisy observability and uncertainty due to stochasticity of the outcome performing an action in a given state.

We deal with uncertainty using “probability theory”.

**Probability**

Several interpretation of probability exist. Two of them we think it is to worth notice here are:

**Frequentists’ Interpretation** Frequentists refer to probabilities when dealing with specific random experiments. “The probability of a random event denotes the relative frequency of occurrence of an experiment’s outcome, when repeating the experiment. Frequentists consider probability to be the relative frequency in the long run of outcomes.” (23)
Bayesians’ Interpretation Bayesian probability describes probability as “a degree of plausibility of a proposition based on the given state of knowledge” (24) in contrast to interpreting it as a frequency of some event.

For the scope of this work, we prefer to use the latter definition of probability.

Several attempts have been made also in giving a formal description of probability, such as Kolmogorov and Cox formalization. We use the first one.

Important definitions are:

The Sample Space $\Omega$, is the set of all possible outcomes of a random trial.

An Event is any subset of the sample space.

Probability values ($f$) are assigned to events ($x$) following these properties (in order to have a consistent probability theory):

- Probability of an event is a null or positive quantity, $f(x) \in [0, 1]$ for all $x \in \Omega$
- Probability of the entire sample space is one, $\sum_{x \in \Omega} f(x) = 1$
- If two events are disjoint, the probability that either of the events happens is the sum of the probabilities that each happens. (If $A \cap B = \emptyset$, $P(A \cup B) = P(A) + P(B)$ (25)

2.2.1 Utility Theory

Having probability theory to capture environment uncertainty, we need a way to capture and describe agent’s preferences. The tool we use to do so is “utility theory” (13).

A utility function $U(s)$ returns a value describing how desirable a state $s$ is.
**Principle of Maximum Expected Utility**

We do not go through details of utility theory that can be found in (13). It is only important to know what “expected utility” of an action $a$, given evidence $e$ is:

$$EU(a|e) = \sum_{s'} P(RESULT(a) = s'|a, e)U(s')$$

I.e., the average utility values of the results of an action, weighted by their probability. And what the “principle of maximum expected utility” (MEU) says about the action that should be taken by a rational agent:

$$action = \arg \max_a EU(a|e)$$

I.e., the action that would maximize the expected utility.

**Multi-Attribute Utility Theory**

Sometimes, it may happen that we have to deal with “multiple attributes” of outcomes to define utility. In this context we define “utility independence” (13) between attributes when values of one attribute do not influence preference of values of another attribute.

If a set of attributes is mutually utility independent (MUI), then we can use a multiplicative utility function:

$$U = k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_1k_3U_1U_3 + k_1k_2k_3U_1U_2U_3$$

is the general case for three attributes (13).
2.3 Complex Decision Making and Planning: Markov Decision Processes

In general, complex decisions are not only taken in uncertain environments. Often the environment is also non-episodic. Problems in such environments are called sequential decision problems and they require to integrate in our rational agents the ability to do planning.

In the context of Markov processes or Markov Chains, is relevant make a Markov assumption. Markov assumption states that “the current state depends only on a finite fixed number of previous states” (13).

The simplest Markov processes are also called first-order Markov processes. They are the ones in which old the - so called - “Markovian property” ,i.e., the current state only depends on the previous one. (Please notice: the environment is stochastic and that is why we are considering probabilities.)

\[ P(X_t|X_{0:t-1}) = P(X_t|X_{t-1}) \]

We can extend this idea in the context of an agent performing actions. In this case we will have to specify a transition function in the form:

\[ P(s'|s, a) \]

If we also assume, or grant, that such transition function does not change over time, i.e., the environment is stationary, then we have a powerful and simple model of our environment.
2.3.1 Elements of an MDP

A Markov Decision Process (26) is a problem defined in a fully observable, stochastic, stationary environment and it is composed by four elements:

A set of States $S$ containing the initial state $s_0$. One or more states can be “final”.

A set of Actions $A(s)$ for each state.

A stochastic transition model $P(s'|s,a)$

A reward function $R(s)$ that returns a value for each state.

Please notice that $S$ and $A$ are sets, $P$ is a probability distribution and $R$ is a function.

2.3.2 Policies

The solution of a Markov Decision Process is:

A Policy $\pi(s)$, a function defined over every state in $S$ that returns an action $a$ to be performed.

The policy must be defined over all the possible states because, due to the stochastic nature of the environment, the agent might, in general, end up in any state at any moment.

The best solution is an “optimal policy”, i.e., the policy with the maximum expected utility.

A “proper policy” is a policy that is granted to eventually reach a final state.
Utility of a Sequence of States

The “performance measure” of an agent in a Markov Decision Process is based on the rewards that the agent collects over the states it explores during its life span.

$$U_h([s_0, s_1, ..., s_n])$$

However, several ways to express this “performance measure” or utility exist.

**Finite Horizon** If we are considering a finite horizon of $N$ steps, it means that any reward collected after the $n$-th step is not relevant.

$$U_h([s_0, s_1, ..., s_{N+k}]) = U_h([s_0, s_1, ..., s_N])$$

In general, optimal policies under finite horizon are non-stationary (the action that should be taken depends not only on the state we are considering but also on the current step index). This means that MDPs with finite horizon are more difficult to treat than infinite horizon MDPs.

**Infinite Horizon - Additive Rewards** We prefer to deal with infinite horizon MDPs, where optimal policy can be written as $\pi(s)$. A simple way to express the utility of a sequence of states is adding the rewards of each state.

$$U_h([s_0, s_1, ...]) = R(s_0) + R(s_1) + ...$$
**Infinite Horizon - Discounted Rewards** A generalization of additive rewards are discounted rewards, where $\gamma$ is a parameter in the interval $[0, 1]$.

$$U_h([s_0, s_1, s_2, ...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$$

Even if additive and discounted rewards appear to be similar, we usually prefer discounted rewards because of two reasons:

- Discounted rewards of an infinite sequence of states are still finite values.

- Additive rewards may cause the failure of optimal policy computation when improper policies exist.

**Utility of a Single State Given a Policy**

Given a policy $\pi$ and an initial state $s$, the amount of reward (or utility) that we expect to collect (starting from that state and applying the policy) is:

$$U^\pi(s) = E[\inf_{t=0}^{\infty} \gamma^t R(S_t)]$$

The optimal policy $\pi^*_s$ for an initial state $s$ is the one that maximizes this utility. As we said before, using discounted rewards and an infinite horizon, the optimal policy is independent from the state that we are considering, so we can just write $\pi^*$.
The true utility of a state is the expected reward under the optimal policy:

\[ U(s) = U^\pi(s) \]

Knowing the utility of each state and the transition model, we can easily compute the action suggested by the optimal policy in each state:

\[ \pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s') \]

Utilities/Optimal Policy Computation

Two very popular algorithms are used to solve Markov Decision Processes. One is known as “value iteration algorithm”, the other one is known as “policy iteration algorithm”. Before going through them, it is useful to recall the “Bellman equation” for the computation of utilities of states:

\[ U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U(s') \]

Value Iteration Algorithm In the value iteration algorithm, utilities of states \( U_i \) are initialized randomly, then they are updated through the rule:

\[ U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s') \]

Optimal policy is computed straightforward after the convergence of the utility values.
**Policy Iteration Algorithm** Policy iteration algorithm starts from a policy $\pi_0$ and it tries to improve it. First, the utilities under the current policy are computed solving the linear system:

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

Then, for each state $s$, if:

$$\max_{a \in A(s)} \sum_{s'} P(s'|s, a)U_i(s') > \sum_{s'} P(s'|s, \pi_i(s))U_i(s')$$

the policy is updated changing the recommended action for that state with the one maximizing the expected utility. The two step procedure is iterated until convergence.

### 2.3.3 Complexity of MDPs

How hard is it, to solve Markov Decision Processes? A detailed and very technical discussion about the all the math behind complexity of these algorithms is beyond the scope of this work. However, some interesting insights can be found in (27). Here we give a brief recap on computational complexity theory and then we review the most relevant findings of that work.

In computational complexity theory, $\mathbf{P}$ is the class of the problems that can be solved in polynomial time with the size of their input.

$\mathbf{NP}$ is the class of the problems that can be non-deterministically solved in polynomial time with the size of their input or, the class of the problems whose solutions can be verified in polynomial time.
Moreover, NP contains P and NP-complete is the class of NP problems to which any other NP problem can be reduced in polynomial time.

The P-complete class is composed by those problems that are in P to which any other P problem can be reduced (28) by some kind of reduction.

Not only time complexity is important, space complexity is relevant too. The PSPACE class is composed by all the problems that can be solved in polynomial space. Notice that the input size is not considered in space complexity, so we can have less than linear space complexity.

P and NP are subsets of PSPACE (27).

“The class NC is the set of all languages L that are decidable in parallel time \((\log n)^{O(1)}\) and processors \(n^{O(1)}\).” (28)

A relevant theorems that is given and demonstrated in (27) and we only report is:

**Theorem**  “The Markov Decision Process problem is P-complete in all three cases (finite horizon, discounted, and average cost).”  (27)

The three cases are the three different way of computing utility of a sequence of states we discussed before. This makes sens if we think of the fact that the algorithms proposed for the solution of MDPs (value iteration and policy iteration) are, respectively, an iterative and an optimization linear programming approach. Both of these approaches are inherently sequential and hard to parallelize (27).
2.4 **Active Reinforcement Learning**

Reinforcement learning is a fascinating discipline that as its roots in “cybernetics and work in statistics, psychology, neuroscience, and computer science” (29). Reinforcement learning carries a huge expectations, it promises to provide “a way of programming agents by reward and punishment without needing to specify how the task is to be achieved” (29).

In (13) we find a precise distinction between “active reinforcement learning” and “passive reinforcement learning”. The latter one, it basically consists of algorithms to learn utilities of states of a Markov decision process when a fixed policy to follow has been specified.

In this work, however, we only stick to the first type of reinforcement learning, the active reinforcement learning, in which an agent “must learn behaviour through trial-and-error interaction with a dynamic environment” (29).

Moreover, we can distinguish reinforcement learning algorithms that perform a search in the space of possible behaviours (this is the case of genetic algorithms and genetic programming) and algorithms that use statistical techniques to estimate the utility of concepts such as states and actions. In this work, we consider the second class of algorithms.

Active reinforcement learning in Markov decision processes is quite completely different from reinforcement learning as it is intended in biology, in (30) reinforcement learning in the human brain is considered in a context where there is no planning and examples are externally labelled.

A general framework for active reinforcement learning is composed by:

\[ S \]

a set of states of the environment.
A, a set of actions the agent can perform.

Scalar reinforcement signals, a binary signal or real numbers, representing a reward.

The agent is placed in the environment, it performs actions that change the state of the environment. When there is a state transition and/or an action is performed, the agent receives immediately a reward/reinforcement signal. Moreover, the agent receives some information about the current state of the environment.

The goal of a reinforcement learning algorithm is to learn a policy that maximizes the long run collection of rewards.

It is worth notice that:

- The agent only receives immediate information.
- There is no presentation of input/output pairs as in supervised learning tasks.

Estimating the performance of a learning algorithm is not a trivial task, e.g., convergence to optimality is interesting information, proved for some algorithms under certain circumstances, but “useless in practical terms” (29) because, even if the optimality is reached at infinity, this does not say how the agent behaves before. Regret is “the expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally from the very beginning” (29), probably the measure we would really want but it is hard to obtain.

2.4.1 Learning a Model and Utilities using Adaptive Dynamic Programming

Now, we present several techniques that can be used to implement active learning agents that are able to solve a Markov decision process and obtain an optimal policy for it. For the
We can use adaptive dynamic programming (ADP) to implement an agent that learns the transition model of the Markov decision process. This is not too hard because of the perfect observability we have assumed for Markov decision processes.

The adaptive dynamic programming agent uses two tables:

- $N_{sa}[s, a]$ is used to keep track of how many times the agent performed action $a$ when in state $s$.
- $N_{s'|sa}[s', s, a]$ is used to keep track of how many times the agent performed action $a$ when in state $s$ and landed in state $s'$.

At each learning iteration, for those state-action pairs who have non-zero $N_{sa}[s, a]$, the estimated transition model is updated by the rule:

$$P'(s'|s, a) \leftarrow \frac{N_{s'|sa}[s', s, a]}{N_{sa}[s, a]}$$

Having an estimated (by maximum likelihood) transition model, utilities of each state can be computed by using the iterative procedure:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a) U_i(s')$$
Or, if we are using a fixed policy, solving the linear system of equation:

\[ U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s))U_i(s') \]

Every time we encounter a new state we initialize its utility with its reward value. We can use an additional table to store the reward values of each state encountered, if we assume that the agent does not have direct access to the reward function.

Even if it usually simple, and sometimes reasonable, to directly choose the optimal policy looking at this estimated model, this is not always the best choice.

We report two approaches that are presented in (13) to choose the optimal policy.

**Bayesian Reinforcement Learning** We assume to know the prior probability of each possible model \( h \), \( P(h) \). We can compute the posterior probability \( P(h|e) \), where \( e \) is what we have observed by now, using Bayes’ theorem:

\[ P(A|B) = \frac{P(B|A)P(A)}{P(B)} \]

If we define the expected utility of a policy \( \pi \) in the model \( h \), averaged over all the possible initial states, as \( u_h^\pi \), the optimal policy would be:

\[ \pi^* = \arg\max_\pi \sum_h P(h|e)u_h^\pi \]
Robust Control Theory In this approach, only a set of “possible” model is considered (e.g. using a likelihood threshold) and the optimal policy is the one that gives the maximum utility in case of having the “worst” model in the set.

\[ \pi^* = \arg \max_{\pi} \min_h u_h^\pi \]

2.4.2 Model-Free Methods

We now discuss three learning algorithms that have their core on an “update rule” used to compute utility values. These algorithms have the property of performing these updates without using a transition model (and so without the need to learn one) of the underlying Markov decision process.

2.4.2.1 Temporal Difference Methods in Passive Learning

In Temporal Difference (TD) learning for a passive agent (one with a given fixed policy), as in adaptive dynamic programming methods, every time we encounter a new state its utility is initialized with its reward value and an additional table to store the reward values.

In this approach, we only use one table of occurrences:

- \( N_s[s] \) is used to keep track of how many times the agent performed visited a state \( s \).

The heart of the algorithm is its update rule:

\[ U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)) \]
It is worth notice that the update only depends on the successor state that is actually encountered. This would lead to an “odd” update when a very rare transition happens, but, being this transition “rare”, the average values of the estimated utilities \( U^\pi(s) \) will converge.

To actually have the values \( U^\pi(s) \) converging to the correct values we must work on the learning rate \( \alpha \). It must be a decreasing function of the times the agent visited the state \( s \) we are considering, such as the one used in (13):

\[
\alpha(n) = \frac{k}{k + n - 1}
\]

### 2.4.2.2 Q-Learning

The approach just presented has a major limitation, it is a passive learning algorithm, it learns utilities of states for a given policy. It is not what we are looking for, we are interested in active agents, able to learn new policies.

Q-learning is a learning techniques with many elements in common with the temporal-difference presented above. Q-learning is a “model-free” approach as well and it does not need to learn the transition model.

In Q-learning, the agent does not learn utilities of states. It learns, instead, the value of performing action \( a \) in state \( s \), these values are called Q-values \( Q(s, a) \). Utility of each state can be derived by Q-values simply using:

\[
U(s) = \max_a Q(s, a)
\]
The update equation when the agent moves from state $s$ to state $s'$ is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Then, to choose its next action, the agent has to decide among exploitation of its current knowledge and exploration. How to do so, it is discussed later.

Notice that, to allow convergence, $\alpha$ is a decreasing function of how many times a state-action pair has been tried: $\alpha(N_{sa}[s, a])$.

Q-learning also has extensions in the multi-agent context (31), typical of some research works about autonomic computing (32).

### 2.4.2.3 SARSA

Similar to Q-learning, SARSA learning (State-Action-Reward-State-Action) has a slightly different update function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$$

The update is performed when the agent already knows it ended up in state $s'$ after performing action $a$ in state $s$ and action $a'$ is the one it took from there (state $s'$).

SARSA is a - so called - on-policy algorithm, because it performs updates actually looking at the policy it is using at that given moment. Q-learning is, instead called an off-policy algorithm.

**Other Approaches**

Many other approaches (such as adaptive heuristic critic, $TD(\lambda)$, prioritized sweeping, etcetera)
and variants of the algorithms we discussed are proposed in texts like (13) and (29) and (33). They are beyond the scope of this work.

2.5 Exploration

Unlike supervised learning algorithms and passive reinforcement algorithms, active reinforcement learning algorithms require exploration of the environment. Exploration is probably one of the hardest problem in active reinforcement learning.

The k-Armed Bandit Problem

The simplest problem of reinforcement learning is the k-armed bandit problem. The environment is composed by a set of $k$ gambling machines (one-armed bandits), the agent can perform $h$ pulls, pulling any arm $i$ of the $k$ available. Rewards of each pull are 0 or 1, according to $p_i$ the (unknown) probability of win of the $i$-th machine. The $p_i$s are independent.

This problem is equivalent of a single state problem with $k$ self-transitions (one for each arm).

In (29), it is suggested how to exploit dynamic programming to calculate the optimal strategy for this problem. The optimal strategy is obtained mapping belief states to actions using their utilities.

Gittins index is another technique that guarantees the optimal exploration and exploitation balance under the condition of discounted rewards (29).

Given the discount factor, the values of the Gittins index are published in tables by values of $n$ and $w$. The action to be choose should be the action $i$ such as that $I(n_i, w_i)$ is the greatest among all the possible actions.
2.5.1 General Multi-State Problems: Exploration Vs. Exploitation

In general, in the multiple state situation, the approaches for the single state problem can be replicated. However, the theoretical guarantees are no longer valid (29).

The learning agent in a Markov decision process has to make choices for which optimal behaviour has not been discovered yet.

The problem that an active learning agent in a Markov decision process has to deal with is the problem of finding the right trade-off between exploration and exploitation.

While the agent explores the environment, it learns some knowledge but how can it know when is the moment (if there is one) to stop exploring and start using this knowledge to collect reward? How can it exploit the partial knowledge it has while exploring not to have a too poor initial performance?

Two important works ((34) and (35)) by Sebastian B. Thrun deal with these issues. They explain how hard the exploration problem really is and they propose a classification (that we follow) for exploration techniques distinguishing between “undirected” and “directed” exploration.

In (34), in particular, Thrun re-proposes a theorem by Whitehead.

**Theorem** In any finite homogeneous problem solving task, the expected time using undirected exploration (such as random walk exploration) required to identify an optimal policy is bounded below by an expression exponential in the depth of the state space \( l \), if the number of actions leading away from the goal state is larger than the number of actions leading closer to it. (36)
This theorem clearly states how tremendously inefficient random and undirected explorations techniques could be in some scenarios.

However, still in (35), Thrun proposes and demonstrate some other theorems (that we do not re-propose here). These theorems show how, under certain circumstances (such as the environment being finite and deterministic), some directed exploration techniques (in particular local counter-based ones) can find optimal policies more efficiently (polynomial complexity with actions and states).

A Generic Exploration Function

A very simple and general example of exploration function is the piecewise-defined function defined in (13). In this function, $u$ is the expected utility computed using the knowledge the agent has learnt so far, $n$ is the times it has visited a given state, $N_e$ a threshold in the number of visits, $R^+$ an optimistic estimation of what reward the agent could collect learning something new.

$$
f(u, n) = \begin{cases} 
R^+ & \text{if } n < N_e \\
u & \text{otherwise}
\end{cases}
$$

To be used in value iteration:

$$
U^+(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} f\left( \sum_{s'} P(s'|s, a) U^+(s'), N(s, a) \right)
$$
This function has a strong limitation in the fact that all the incentive in performing exploration is at the beginning of the agent life span. This is, for example, absolutely inefficient and misleading in the case the environment was not stationary.

2.5.1.1 Undirected Techniques

In (34), Thrun describes several techniques for both direct and undirect exploration that here we briefly summarize.

In undirected exploration, exploratory action are chosen using some basically randomized criterion. No additional knowledge is stored or used to perform the exploratory task. The techniques we found in literature are described below.

Random Exploration In random exploration, the next exploration action that the active learning agent will perform is chosen using a uniform probability distribution over all the possible actions in the current state. This technique is terribly inefficient if we look at the fact that it is completely unable to exploit any of the knowledge the agent has during the exploration phase. Furthermore, it does not move to knowledge exploitation at any point, we should explicitly state when the agent will have to switch to the exploitation phase.

It might make sense to use this technique when the exploration/learning phase and the exploitation phase are clearly separated.

Utility-Driven Probability Distributions If we want to exploit the current knowledge during learning and exploration, the easiest way to do so is a modified version of random exploration.
Instead of using a “uniform probability distribution over all the possible actions in the current state”, the agent should use a probability distribution in which the actions that are known (or believed) to lead towards higher utilities have a greater probability to be chosen.

**An Example of Semi-Uniform Distributions** In (34), an exploration rule with a probability distribution is proposed. It relies on what Thrun defines as “exploitation measure” of an action in a state:

\[
f_s(a) = \sum_{s' \in S} P(s'|s, a) \hat{U}(s')
\]

The action that maximizes this measure is the one that should be more likely to be chosen, so the probabilities of choosing each of the action available in the current state are computed as:

\[
P(a) = \begin{cases} 
P_{\text{best action}} & \text{if } a \text{ maximizes } f \\
\frac{1-P_{\text{best action}}}{\# \text{ of actions}} & \text{otherwise}
\end{cases}
\]

Where \(P_{\text{best action}}\) is a parameter from pure randomness (if zero) to pure exploitation (if 1).

**Boltzmann Distributions** “Boltzmann-distributed exploration takes the f-evaluations of all actions into account. The probability for an action to get selected is:

\[
P(a) = \frac{e^{f(a)/\theta}}{\sum_{a' \in A} e^{f(a')/\theta}}
\]
Here $\theta$ is a gain factor, often called temperature, which determines the amount of randomness in the action selection procedure. With $\theta \to 0$ pure exploitation is approached and with $\theta \to 1$ the resulting distribution approaches the uniform distribution, i.e., random exploration.” (34)

2.5.1.2 Directed Techniques

Directed exploration techniques are, according to the definition given in (34), those techniques that exploit “ad hoc” knowledge such as counters and recency information.

Counter-Based Exploration Counter-based exploration techniques exploit the information about how many times states have occurred. An example of counter based technique proposed in (34) is:

$$eval(a) = \alpha f(a) + \beta \frac{c(s)}{E[c(s')]}$$

Here, the exploitation measure $f$ is linearly combined with a term that is increasing with the counter of the current state and decreasing with the counter of the expected state the agent will end up is it perform the considered action.

Counter-Based Exploration with Decay A way to improve counter-based techniques is, not just to consider the frequency information contained in counters, but to add information about “when” actions have been performed/states have been explored. Usually, if a states has been visited many times but a lot of time ago, the agent should be willing to explore it again.
We can easily include this information making counters decay over time:

\[ c(s) \leftarrow \lambda c(s), \forall s, \lambda \leq 1 \]

**Using a Recency Measure** Instead of using decay, we could define a “recency measure” for a state \( \rho(s) \) as Thurn does. A possible evaluation criterion for actions using this measure would be:

\[
\text{eval}(a) = \alpha f(a) + \beta (E[\rho(s')])^{1/n}
\]

The term \((E[\rho(s')])^{1/2}\) is called “exploration bonus” by Sutton (37).

**Error-/Counter-Based Exploration** An other way to take into consideration “what happened lately” and obtain a more complex counter based criterion for choosing the next action is to store information about the last change of the utility estimate for each state \( \Delta U(s) \).

The basic counter-based criterion becomes:

\[
\text{eval}(a) = \alpha f(a) + \beta \frac{c(s)}{E[c(s')]} + \gamma E[\Delta U(s')]
\]

Please notice how, in general, the actual computation of all the terms containing an “expected value” \( E \) would need the knowledge of the transition model. In a real implementation we would use an estimate of it.
More Advanced Techniques

Exploration and exploitation are both necessary but they usually lead the agent behaviour in opposite directions. Having a linear combination of the two as in:

\[
eval(a) = \alpha f(a) + \beta \frac{c(s)}{E[c(s')]} \]

might not be a great idea. Put together, one could nullify the other. In (35) and (34), Thrun suggest the use of a parameter that he calls “attention”.

\[
eval(a) = att \cdot f(a) + (1 - att) \cdot \frac{c(s)}{E[c(s')]} \]

We do not go through the way he suggests to use for the updates of this value. However, it is clear how modifying the attention over time we can influence the focus of agent on exploration or exploitation.
In The Following

So far we have presented several types of environments, rational agents, utility theory, complex decision making and optimal planning through Markov decision processes, application of reinforcement learning to Markov decision processes and some exploration strategies that could be used when learning has to happen on-line.

In the following of this work we will see that (some aspects of) an operating system can be translate into an environment/rational agent pair. Then we will formalize a Markov decision process out of this computing system, specifying states, actions and rewards. Finally, we will implement an agent (controller) able to solve on-line this Markov decision process exploiting reinforcement learning.
CHAPTER 3

RELATED WORK

“Alcohol is for people who can afford to lose some brain cells.”

Charlie Sheen (Two and a Half Men)

This chapter presents and summarize papers, articles and others published texts that deal with the main aspects of this research work.

We start presenting works whose focus is on fundamental properties of autonomic computing systems, such as self-awareness (in the first section) and being self-adaptive (in the second section). Then, we discuss an architecture previously proposed to solve the same problem we are facing. Finally, we extend the discussion to the subjects of artificial intelligence and learning (in the context of autonomic computing systems), as well as to other open issues in this field.

3.1 Monitors: Making Systems Self- (and Context-) Aware

As stated in (1), (11) and (12), one of the fundamental properties of an autonomic computing system is the property of being self-aware.

In order to be self-aware (and environment-aware), the autonomic element must be able to gather information about its internal state and the state of the environment it is working in. The property of self-awareness is usually achieved instrumenting the autonomic element with monitors, or sensors, if we want to use the terminology of artificial intelligence (13).
To avoid misunderstandings, it is important to notice that the distinction between self- and context-awareness does not depend much on the monitors but only on where we draw the imaginary line between the environment and the autonomic agent.

In general, a single monitor might be able to provide information about only one of the internal state or the surrounding environment. This means multiple monitors may be necessary and desirable to obtain self- and context-awareness.

The importance of having the ability to collect (and manage) knowledge and the related difficulties are strongly remarked in (38):

“The paper rejects both the assumed universality of tacit-explicit conversion and recent arguments that the phrase knowledge management is an oxymoron. This is achieved by embracing the paradoxical nature of knowledge as both a thing and a flow.”

In (11), Sterritt and Bustard present two different monitors that could open the way to self-awareness: the Heart Beat monitor, a general name for monitors able to collect simple but still useful information about applications (they, however, state that more complex monitors might be needed to detect “how good” an application is performing) and the NASAs Beacon Monitor (39) (40), a monitor that allows spacecrafts to send signals to the Earth containing the information on how important it is to track the position of the spacecraft.

Speaking of the ability of the system of monitoring the surrounding environment, it is common to use or find the term “context-awareness”. In the field of “context awareness” we cite works such as the CyberDesk system proposed in (41), a software architecture for context-awareness. In (41) context-awareness is defined as including, but not limited to, “information
the user is attending to, emotional state, focus of attention, location and orientation, date and
time of day, objects and people in the users environment". In (42), from University of California,
Berkeley, context-awareness is explored both in general and in the real implementation of the
“Context Toolkit”. This work identifies different benefits (such as “allowing sensors and services
to be upgraded independently of one another and dynamically while the system is still running”)
and challenges (such as “scoping of sensor and context data to ensure security and privacy”) of
this context toolkit.

Two important works, strongly related to this one are (3) and (5). (3), from MIT, presents
Application Heartbeat and it raises the point that, often, applications are “performance black-
boxes and adaptive services must infer application performance from low-level information or
rely on system-specific ad hoc methods” (3).

Application Heartbeat is suggested as a way to overcome this problem:

“The Application Heartbeats framework is designed around the well-known idea of a heart-
beat. At important points in the program, the application registers a heartbeat. In addition,
the interface allows applications to express their performance in terms of a desired heart rate
and/or a desired latency between specially tagged heartbeats. [...] Thus, Heartbeat-enabled
applications are no longer performance black-boxes.” (3)

Application Heartbeat has a “simple, conventional programming style” and it “uses only
standard function calls and does not rely on complex mechanisms such as OS callbacks.”
The goal of Application Heartbeat is to provide a “unified, portable standard for application performance monitoring”, which is considered a mandatory step for the successful realization of autonomic computing systems of the future.

The most relevant features of application heartbeat are:

- It provides an API for monitoring.
- It is intended to monitor the throughput of loop-based application.
- It is implemented in user-space.
- A process is responsible for collecting statistics.
- Communication between the monitoring process and monitored application is implemented through shared memory or files.
- Targets are specified to the monitor process as minimum and maximum heart rate.
Heart Rate Monitor (HRM) (5), developed in Micro-Architectures Laboratory (uLab) in Politecnico of Milan, is a performance monitor for self-adaptive computing. Just like Application Heartbeat, it provides a general way to instrument applications and measure their performance. Notice that the applications considered in this research are defined as “iterative” and “rate-based”. Applications in which completion time and latency are the main concerns are not directly addressed by this work.

Heart Rate Monitor has been developed looking at the open source Application Heartbeats and with the precise goal of improving its performance and functionality.

Application Heartbeats has many qualities, such as usability and portability. HRM differs from Application Heartbeat by the fact of being “partitioned between user- and kernel-space” and being “integrated with Linux”. Heart Rate Monitor sacrifices “portability to functionality while maintaining a simple API”.

Another aspect in which HRM differs (and improves) from Application Heartbeat is the capability of working, not only with processes and their identifiers (PIDs), but also with finer grained execution units such as threads (identified in Linux by unique TIDs) and more complex execution conglomerates, such as groups (subset of of an application that might be single-threaded, multi-threaded, multi-processed, or any feasible combination of them).

### 3.2 Decisions and Actions: Making Systems Self-Adaptive

A self-aware system is still far the broader vision of autonomic computing presented in (1). Only to complete the observe-decide-act control loop and obtain self-adaptivity, the autonomic element must be able to make some kind of decision and ii must be able to perform them.
Self-adaptivity has been explored, in literature, in many aspects and at very different scales, from the algorithm/data level in (43), where an intelligent manager “automates method selection based on data, algorithm and system attributes” to the dimension of operating systems as in (44), a paper of Margo Seltzer and Christopher Small from Harvard University that dates back in 1997.

It is not trivial to notice how, while monitoring and self-awareness are usually considered as a topic with its own, stand alone, relevance, typically, when we consider self-adaptivity, the separation in between the decision making underlying the choice of an action and the set of all possible actions is blurred or implicitly stated.

Starting from this consideration, we consider here three different works, produced one after the other, that explore the implementation of self-adaptivity over the self-aware framework of Application Heartbeat:

The work proposed in (10) is focused on the application of control theory in order to make systems self-adaptive.

It is stated that very often, if not most of the times, the adaptivity of a system can be obtained directly by re-formulating the problem as a control problem: “consider the problem of allocating processor cores in a modern multi-core processor. A possible strategy is to have the system allocate the minimum number of cores required to meet application goals.” (10)

Monitoring of the system is enabled through the Application Heartbeat software framework previously discussed. The framework performance of applications can be measured and the desired performance values (in the form of heart rates) can be specified.
“For example, a video encoder can be written to produce a heartbeat with every frame of video. Furthermore, this encoder can express its target performance as a desired rate of frames per second.” (10)

Resolving the adaptivity problem as a control problem is strictly model-based: “the successful formulation of computing system control problems relies on a clear definition of the system to be controlled prior to the control design—a less trivial task than it may appear” (10).

The case study of this work is the x264 video encoder with different input data, a variety of experiments and their successful results are also presented in the work.

In (9), a “Framework for Self-aware Management of Multicore Resources”, SEEC is presented. This work draws some of the aspects of the one discussed above. It still deals with complex systems in which the main challenge is the distribution of computational resources of a modern multi-core architecture. Again, the self-awareness is obtained through the monitor functionality of the Application Heartbeat framework.

At least two aspects of this work must be underlined here. First of all, it stresses the importance of modularity.

Many (most) of the self-adaptive systems are based on an observe-decide-act loop (9). In some systems this adaptivity is implemented at the application layer, leaving to the software developer full power as well as full responsibility. In some other systems, adaptivity resides at system/hardware level.

Both of these approaches have the same limitations in the sense that only one scope exists for observation, decision and action.
“In contrast, the SEEC model decouples the specification of observation, action, and decision steps by establishing separate interfaces for these phases. This separation allows different individuals working at different layers of the system to concentrate on the most appropriate phase for their expertise” (9)

Moreover, SEEC can “control the PARSEC (8) benchmarks through resource allocation” (9), and this is (almost) the same benchmark suite that will be used in this work.

In (4), we can find many elements from (10) and (9). Once again, the self-awareness is provided by the Application Heartbeat monitor, as in both (10) and (9). Like in (9) the experimental results are collected on the PARSEC benchmark suite (8). From (9) is derived the strong modular division in the three components of self-adaptive systems, observe, decide and act.

The main element of novelty is that, unlike (10), where only one simple manager derived from control theory was used, in (4) a large set of different managers/controllers/policies is proposed and implement.

All of these policies share the same way to observe and obtain self-awareness, Application Heartbeat. All of these policies operates on the same parameters to make the system self-adaptive (once again, the distribution of computational resources in a multi-core architecture).

The implemented and compared policies in this work go from heuristic solutions to standard and advanced control-based solutions (such as proportional-integral PI controllers or model predictive control MPC).
3.3 The Previous AcOS Architecture

Attempts to introduce the application heartbeat monitoring API in a broader context of self-adapting and self-optimizing systems are in (2), (9), (4).

All of these works pioneered what we now call AcOS. AcOS is the idea of autonomic operating system being developed at uLab in Politecnico of Milan (PoliMi) in the last few years.

These works provide implementations of systems in which, while one or more HB-instrumented applications are running, the information collected in the application heartbeat monitor is used to “adapt” some system parameters in order to “optimize” some measure (e.g. application heart rate, system temperature).

Here, we sketch the global picture of the architecture in which these works were included.

**Services** The original architecture in (2) was, essentially, a multi- (two-)layered architecture.

The bottom level of self-adaptation was provided by - so called - “services”. An abstract definition of a service could be: an active element in the user-space (e.g. a process) that exploits the information collected by a sensor to modify at run-time some parameters at system-scope in order to minimize a distance measure from a fixed goal.

In all the actual implementations, application heartbeat was the sensor and the goal was defined straightforward as the vector of heart rates of the instrumented applications running in the system.
The actuating part was provided by lines of code directly inserted in a service (e.g. the “taskset” command (45) to influence scheduling of PIDs on a specific subset of cores or adjusting niceness of a process (2)).

Several services, differing in the adapted system parameters and in the policy used to compute their new values) are implemented and compared in (4).

**Service Coordinator** In (2), the level above service-level was embodied by a - so called - “Consensus Object”, a generic coordinator of services.

This element of the system was able to access the same information used by each service (e.g. application heartbeat heat rates) and their “goals” and decide at run-time which was the best service to be used in any given situation.

Several implementation of the decision algorithm for this element are proposed in (2), such as heuristics and machine learning-based ones.

This level of coordination enables self-optimization (and consequently a partial self-management) of the system at a higher level, providing a more evolved implementation of autonomicity in the system.

**Limits of This Approach** The two-layered structure presented so far achieved very good results, especially when only considering a service controlling one application heart rate, that are shown in (4)
Figure 7. Representation of the original architecture, from “Application heartbeats: a technique for enhancing system self-adaptability” (2010)
However, it suffers of a few architectural limitations and it fails in exploring some possibilities of further enhancing aspects of autonomic computing such as the ability to freely self-configure.

**In the Practical Implementation** The first obvious limitation in the described implementation is the lack of the ability to self-configure for generic sensors and actuators. Think of each service as an intelligent agent performing the task of self-optimization of the system (environment of the agent), the agent function, the mapping from states described by sensors values to the actions, it is fixed at compile-time.

We might introduce new services but, in general, this should be done off-line. Moreover, this implementation does not allow more than one service at a time to operate on the same system parameters (for obvious reason that one service could nullify the other one).

**In the Theoretical Approach** The theoretical limits of this approach are a direct consequence of the practical limits described above.

Even if (4) makes an attempt to introduce machine-learning at service level (i.e. as a stand-alone policy with fixed sensors and actuators) most of the policies used so far in works such as (9) and (2) were simple heuristics or control theory-based approaches.

Missing the opportunity of using advanced artificial intelligence techniques (46) (such as learning and specifically reinforcement learning), we are missing the opportunity to create systems able to naturally adapt to multiple situations (thinking not only
of how the environment is or it behaves but also what is currently in the capabilities of the agent in term of sensing and acting).

In particular, we are giving away a feature of reinforcement learning that seems to be essential for truly autonomic computing: the ability, for the agent, to make planning, even when only high level tasks are given, without further details.

3.4 Artificial Intelligence in Autonomic Computing

In (6), Jeffrey Kephart and William Walsh of the IBM Thomas J. Watson Research Center give an extended analysis of the possible applications of artificial intelligence in autonomic computing.

Kephart and Walsh characterize autonomic systems as they are characterized in artificial intelligence, introducing the concepts of state, action, probabilistic transition function, etcetera.

Despite the fact that many different works already attempted to give a definition of policy, drawing from various field of science (47) (48), Kephart and Walsh use a very broad definition of policy: “a policy is any type of formal behavioral guide” (6). Then, they approach policies as rational agents are presented in (13). They concentrate especially on “reflex”, “goal-based” and “utility-based” agents. They do not take in account learning agents that are present in (13) and explored in the context of autonomic computing by, for example, (7).

Action policies are the basic component of reflex agents. In (6) reflex agents are considered essential for the starting of autonomic revolution (or evolution (12)) but it is stated that true self-management, primary goal of autonomic computing systems, could be achieved only through higher level goal policies and utility function policies.
Goal policies differ from action policies in the fact that they do not directly specify an action to take in each state. They, instead, specify one or more “desired states”. Goal policies have the advantage of freeing “human policy-makers from the necessity of knowing low-level details of system function” (6) but also some drawbacks such as the difficulty to express preferences among desired goal states or the necessity of complex and sophisticated planning and modeling algorithms.

In (46), while stressing the importance of thinking of autonomic systems as composed by intelligent agents, Kephart says: “Industry work on autonomic computing has typically framed autonomic elements as services rather than agents”.

“Utility-function policies are much more appropriate for autonomic computing than action policies because they focus on desired state. (49)

Utility function policies “generalize Goal policies [...] In situations in which multiple Goal policies would conflict (i.e. they could not be simultaneously achieved), Utility Function Policies allow for unambiguous, rational decision making” (6).

Most of the times, utility function policies introduce the added technical and computational complexity of requiring optimization algorithms.

The ideas of using utility theory (a widely used instrument in artificial intelligence problem specification) has been discussed in many texts, such as (49), (50), (51), (52) and (53), the latter one, in particular, focuses on the introduction of these tools in actual commercial IBM products: WebSphere Extended Deployment and Tivoli Intelligent Orchestrator.
All of the policies proposed and implemented in (6) are tested on virtual system representing a data center scenario. In this scenario, multiple application managers work simultaneously, each of them with its own pool of identical servers. Each application manager implements a policy as defined in a global “policy repository”. Moreover, a global “resource arbiter”, with its own policy, is responsible for assigning more resources to each application manager when requested and/or available.

This scenario, sometimes referred as “Unity”, is recurrent in many IBM research projects on the application of artificial intelligence in autonomic computing, (53), (49), (54), (55) and others. Probably the more detailed description can be found in (32).

### 3.4.1 Mixing Policies

In the experimental results of (6), it is worth notice how different components can implement different policies. This observation, however, does not lead to the degree of freedom we could expect or desire: “policy types cannot be mixed arbitrarily. For example, the Arbiters policy has to be consistent with the form of the resource requests, which depend on the type of policy used by the AM issuing the request” (6).

Moreover, at a lower level, policies of different type can be mixed in the same component of the system. In particular, (6) says it is quite easy to mix goal and utility-based policies. This is due to the fact that the latter type is a generalization of the former: “combining Utility Function and Goal policies within a single component will involve using a Goal policy as a constraint on the optimization problem defined by the Utility Function policy” (6).
This means, in general, that mixed goal-utility policies will have the same limitations of simple goal policies.

The combination of action policies with different (and more complex policies) appears to be much more challenging, in (6), two possible solutions are suggested: “ensure that they are applied to very different classes of service attributes, such as performance and availability or security” or “ensure that the optimizer cannot control any variable mentioned in the Action clause of an Action policy”.

A final remark has to be done about mixing policies: because of what has been presented, we can say that, contrary to popular believing, “Policy Sets are the right way to think about policy” (6) and the typical approach of considering only policies as stand alone and not thinking about their relationships can lead to unexpected and disappointing results.

3.5 Reinforcement Learning in Autonomic Computing

Autonomic computing systems, as proposed in (1), usually relies on autonomic manager that uses some knowledge about the system architecture that many researchers prefer to introduce at design-time. Too bad, this approach has several limitations.

Defining an explicit system model is not always an easy task, and this is one of the reasons limiting the developing of autonomic systems today.

"A key factor limiting rapid adoption and wide usage of self-* managing systems [...] is the difficulty of engineering a sufficiently accurate knowledge module that can achieve acceptable performance in deployed systems. Because todays computing systems are highly complex and
distributed, developing accurate models of them is a potentially complex and time-consuming task.” (7)

Machine learning is a promising technique that could overcome this “knowledge bottleneck”. Machine learning approaches can learn with no (or very little) built-in previous knowledge, as well as “gracefully incorporate any available initial domain knowledge” (7).

A relevant distinction made in (7) that is worth notice in between reinforcement learning policies and the general monitor-analyze-plan-execute control loop is that reinforcement learning planning is known as “reactive planning”, decisions are made “fairly immediately”, while MAPE’s planning includes more general way of creating and performing plans.

“However, RLs reactive planning is quite sophisticated and might be perfectly adequate for many problems in which generative planners are currently used or envisioned.” (7)

A preliminary exploration by Gerald Tesauro of the use of reinforcement learning for on-line resource allocation was in (56).

Tesauro also proved how reinforcement learning could be used in autonomic managers to obtain specified power-performance trade-offs (22), (57).

3.5.1 Challenges

In (7), Gerald Tesauro identifies four main issues for the application of reinforcement learning in autonomic computing systems.

- The need to observe a very large number of state-action couples.
- On-line training will lead to very poor performance as long as the optimal (or a good) policy has not been learnt yet.
• Exploration might be needed and exploration is an expensive and difficult issue.

• Real systems might not be fully Markovian and include partial observability.

The simulation framework used for testing of reinforcement learning approaches in (7) and (56) is the same IBM framework presented in (6). It contains multiple servers, application managers and a global resource arbiter.

In (7), two different types of reinforcement learning are tested: a learning algorithm for Q (utility) values of state-action pairs, it is shown that this approach “can obtain comparable performance to the best-practice approach for on-line resource allocation; a second approach consists in hybrid reinforcement learning, in this approach the learning algorithm can exploit existing knowledge or built-in heuristics even without directly interfacing them.

More details about “hybrid reinforcement learning” can be found in (58), (59), (60) and (61). Hybrid RL mainly differs from traditional RL in the sense that learning is performed off-line while an external model/policy is applied and the learnt policy is used only when convergence is reached.

“This approach completely avoids poor performance [...] that could occur with live on-line training.” (7) “Hybrid RL consistently achieves substantial performance improvements over a wide variety of open and closed queuing network models” (7)

In the end, (7) contains an optimistic note about the application of reinforcement learning in real (and specifically autonomic) computing systems: “we might have expected and feared that real computing systems would prove too challenging for RL: training times could be exorbitant, state descriptions might be intractably complex, and suboptimal performance while learning
and exploring could be prohibitively costly. Although all these factors might still arise, the early results from the case studies are promising and suggest that these factors are perhaps not as bad as expected” (7).

Other relevant works that show the potentiality of reinforcement learning applications in enabling self-adapting systems are: (62), where David Vengerov and Nikolai Iakovlev of Sun Microsystems use on-line reinforcement learning to learn how to distribute computing units of a multi-core processor among two partitions of a Solaris system while the number of threads running on each partition is changing over time; (63), where again David Vengerov uses reinforcement learning to implement (in simulation) a scheduler for real-time systems; (64), where Shimon Whiteson and Peter Stone of the University of Texas use reinforcement learning to learn a routing policy in a simulated multi-tier system.

Interesting examples of the use of artificial intelligence and reinforcement learning in systems that are partially (the scope is limited to power management) self-managed (and, therefore, self-aware and self-adaptive) are given in (65) and (66).

Both of these works draw their motivation from the open issue of power consumption in modern computer systems. High power consumption is responsible for high cooling costs in large high-end computer systems and for short battery duration in portable mobile devices.

A possible solution for undesired high power consumption could be found in dynamic power management (DPM). DPM “selectively shuts-off or slows-down system components that are idle or underutilized” (65). Obviously, dynamic power management requires a smart manager, responsible for the decisions to put in effect.
Most of the power management policies separates system modeling and policy optimization (65), relying, for the modeling part on tools like regression functions or stochastic modeling.

The power manager proposed in (65) and (66), instead, have in common the properties of being model-free and being learnt through reinforcement-learning.

In (65), a modified Q-learning algorithm is used to compute the values of any given state-action couple (i.e. the expected reward of taking an action in a given state). This implies that the computing systems itself has been previously formalized as a Markov decision process (MDP).

In (66), temporal difference (TD) learning is used instead of Q-learning. Moreover, the computing system is considered as a semi-Markov decision process (semi-MDP) and a naive Bayesian classifier is used as a workload predictor.

Both (65) and (66) provide results for experiments performed on real-system as well as simulated systems.

### 3.6 Other Related Work

It a shared belief that technical challenges will arise while autonomic computing systems mature. In (1), they are subdivided looking at their “scope”: autonomic element scope, autonomic element interactions (the idea of autonomic systems as multi-agent system is well represented by the work in (67), the possibility of performing “decentralized resource allocation” in autonomic computing systems has been investigated in (68)), system scope and system/user interaction.
Another IBM paper about autonomic computing challenges is (69) by Jeffrey O. Kephart. Here Kephart divides these challenges in three categories: autonomic element level, autonomic system level and human interaction level.

The possibility of actually implement multi-vendor autonomic computing systems is discussed in (70), examples of autonomic computing in commercial products are in (53).

By now, we have introduced autonomic computing systems and presented some works aiming to their implementation through the realization of self-awareness and self-adaptivity. Then we have discussed of the role of artificial intelligence and reinforcement learning in the developing of autonomic computing systems.

The underlying idea proposed in these pages is that we can realize autonomic computing systems by introducing self-awareness and then implementing an observe-decide-act to make the system self-adaptive.

We also suggested that, even if control theory and other approaches have proved to be effective, an interesting and promising way to go would be the introduction of artificial intelligence (and especially learning agents) in the decide phase of the control loop.

This stated, we do not have to forget that autonomic computing is still a much broader concept, including objectives like the abilities of the system to self-heal and self-protect.

In (71), AdaptGuard is introduced. AdaptGuard is a service for adaptive systems that must cope with instability. This paper addresses the necessity for fault-tolerance, fault-detection and fault-recovery, while pointing out how a model-based decisor (such as the basic control theory
ones) could be found to be ineffective when they “implicitly assume a model of system behavior that may be violated” (71).

Autonomic computing systems have been matter of discussion from quite some time now and the field is gradually becoming a mature one, overcoming initial enthusiasm and later criticism. In recent works we can already find surveys of large sets of autonomic computing related research. In (72), several ideas of self-management (more general than IBM vision) are discussed, works are classified after the aspect of the MAPE control loop principally developed and the fascinating idea of “degree of autonomicity” is introduced. In (73), another survey is presented, classifying works by the sub-objectives of self-management (self-configuration, self-optimization, etcetera) achieved.

Moreover, the idea of “personal autonomic computing” that is proposed in (74), somehow relates to the work proposed in these pages, where concepts of autonomic computing are implemented as desktop-level, creating novelty with respect to most of the examples proposed in the early publications by IBM.

We also have to consider the fact that, despite its popularity and success among the scientific community, autonomic computing is not the only approach it has been suggested to deal with those issues that today limit the growth and evolution of computing systems.

Intel researchers supported the introduction and definition of proactive systems (75). While autonomic computing systems are based on the known four principles of being self-aware, self-adaptive, self-healing and self-protecting, “Proactive system design is guided by seven underlying principles: connecting with the physical world, deep networking, macro-processing, dealing
with uncertainty, anticipation, closing the control loop, and making systems personal” (75). Proactive systems give “emphasis on human-supervised systems, rather than [...] completely automatic systems” (75). However, “there is considerable intellectual overlap between research into autonomic and proactive systems.” (75).
TABLE I

STATE OF THE ART APPROACHES COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>Application Heartbeat (3)</th>
<th>Services &amp; Consensus Object (2)</th>
<th>Comparison of Control Techniques (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presence of Multiple Policies</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Coordination of Policies</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Application Heartbeat</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Heart Rate Monitor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observe/Decide Separation</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Decide/Act Separation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning Capabilities</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible to New Policies</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Extensible to New Sensors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible to New Actuators</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here we present the vision of the system whose implementation will be discussed in the next section and that will be tested in the following one.

This work is about the implementation of a modular and extensible framework for enabling application of intelligent (learning included) resource allocation at operating system level.

It deals with the concepts of monitoring (or sensing), self-awareness, self-adaptation and, generally, with self-management of computer systems. Consequently, it places itself in a broader vision, the vision of autonomic computing, and it focuses especially on making an autonomic element out of a desktop operating system.

In the first section of this chapter we present the ideas behind the AcOS project, then we describe the architecture of the system we are proposing. In the third section we discuss about the most challenging aspect of our work, the introduction of artificial intelligence and reinforcement learning, finally we explain in what this work differs from the previous ones and why it is valuable.
4.1 AcOS: Autonomic Operating System

AcOS is the name we give to the vision of an autonomic operating system it is being developed at uLab in Politecnico of Milan (PoliMi).

The idea of autonomic computing, as proposed by IBM researchers, aims to free end-users from almost every bargain if not high-level goals specification and involves every level of computation.

In a fully developed autonomic environment, computer systems are composed by elements of different granularities, each one of them is an autonomic element, able of self-management (that, itself, is composed by self-configuration, self-optimization, self-healing, self-protection), with minimum participation by the user. Moreover, autonomicity in the general sense arises from the interaction among autonomic elements (76).

AcOS is an attempt to bring autonomicity at the desktop operating system level. Many of the multiple branches of the AcOS project, such as (77), are beyond the scope of this work.

AcOS is Linux-based operating system.

4.1.1 How this Work Fits into AcOS

This work mainly deals with the introduction of self-management (in particular self-configuration and self-optimization) while looking at the operating system as an autonomic element. This element is inclusive of all the user applications running in the OS.

In general, we will consider a monitor (or a sensor, using indifferently the terminology from early autonomic computing (1) and artificial intelligence (13)) anything that can be used to
recollect statistics and information about the current running conditions of the operating system and the applications running in it.

In a similar way, actuator is anything that provides a way to act on the operating system (or on an application) in order to modify its working condition.

Decision making (the “decide” phase or the “analyze-plan” phase of the typical decision loops), in this particular work, draws a whole bunch of techniques from the artificial intelligence field and it is placed into the user-space of the autonomic operating system.

4.1.2 HRM

HRM, or heart rate monitor (5), is a work brought on by uLab of Politecnico of Milan and it can be considered as a new and improved version of the old application heartbeat.

The main new features of HRM are:

- It is implemented in kernel space.
- Statistics are kept in kernel space but they can be made visible to user-space processes though shared memory.
- Statistics no longer refer only to processes but also to threads and “groups”.
- Groups are non-intersecting sets of threads and processes.

“Each group is characterized by its counter, which records heartbeats, statistics, which contain the heart rate (i.e., the frequency at which tasks emit heartbeats), and target, which is expressed as a heart rate range.” (5)

HRM provide two simple API calls to instrument processes:
• *heartbeat* to register an heartbeat.

• *heartbeatN* to register multiple heartbeats at once.

HRM is the monitor of choice of this work.

## 4.2 Proposed Architecture

The architecture proposed in this work wants to represent a step forward with respect to what has been done by now, e.g., in (2).

We want to keep (and put together) all of the features of the older implementations and add new capabilities to them.

The new system we propose is a collection of software elements that work together in order to make an autonomic element out of the operating system they are running on.

By the expression “autonomic element”, we mean a software systems that is able of self-management through complex behaviour when only few, simple goals or desired properties of the system are specified by a human user.

Specifically, by self-management we mean that the elements of this system can interface each other and communicate with little or no external effort (self-configuration) and they act collectively in order to reach the user-defined goals (self-optimization).

As a matter of fact, the “goals” we refer to are specified for the system by human user with a decent level of IT insight: for example, we take into account the (minimum and maximum) heart rates of each running application specified by the developer/coder who instrumented it with the HRM calls, the maximum temperature of the processing units requested by the system administrator, etcetera.
4.2.1 **Modularity**

The previous work composed by “services” and “consensus object” could be seen as a two-tier architecture.

In this work, we propose an architecture preserving this layered decision structures but also adding strong modular separations across the whole system.

We want to preserve the two-level decision system, in which low level mechanisms decide how to map from “percepts” (as in a simple reflex agent, but in general “states”) to actions, and a unique high level arbiter decides which of the lower decision systems to enable or not.

However, to fully enforce the artificial intelligence approach of this work, even in this pre-existing structure, we want to superimpose the observe-decide-act loop modularity that so closely resemble a rational agent structure.

4.2.2 **Acting**

We want to isolate the acting phase. Acting, as we have seen, basically consists of tuning system-level parameters.

We already touched on some examples of acting, such as tuning of affinity masks from processes to cores, niceness, working frequency of cores, etcetera.

In general, not every system allows to tune the same set of parameters using the exact same procedures, depending on hardware or software limitations: sometimes we will only be able to modify the working frequency of all the cores at one instead of choosing a specific value for each core, some other times the “taskset” command will not not the right or best way to influence scheduling of processes/threads.
This is (one of the reasons, the more intuitive one) why we prefer to encapsulate the acting phase in libraries, each one representing an “actuator” or “executor” after the AI terminology.

Actuator libraries provides functions to be used by rational agent software elements with two specific objectives:

- Mask very low level, system-dependent implementation details.
- Providing “cardinality of” and an “iterator over” the set of actions they represent, to be used by artificial intelligence learning algorithms.

4.2.3 Sensing

As for the acting phase, we also want to isolate the sensing/monitoring phase. What we might be monitoring, in general, is any information about the current running conditions of the operating system, the hardware underlying or any application in it.

We said how the previous works mainly relied on application heartbeat for the sensing phase, recollecting heat rates and other statistics for each application heartbeat instrumented process. Similarly, we use HRM that provides a more performing implementation (5) and additional information about single threads, processes and groups. Moreover, we are willing to take into consideration the temperature of the processing units, wherever hardware and operating system allow to do that.

Doubts about effectiveness of such minimal state descriptions are addressed by the optimistic empirical results in (52).

Again, a library is the practical way we use to encapsulate a sensor in order to mask very low level, system-dependent implementation details.
4.2.4 Decision Making

Decision making was distributed over two levels in the original implementation, so it is now. We introduce a slightly different terminology and some new features that are presented in more detail later.

The higher level of decision making, the role previously served by the service coordinator or consensus object is now overtaken by the adaptation manager (AdaM).

Each one of the lower level decision systems is substituted by an adaptation policy (AP).

Adaptation Policies: APs From a logical point of view, each policy is exactly what an agent function is in the artificial intelligence field: a mapping from states (that in the simpler case can be represented by vectors of monitored values) to actions.

These agent functions, just as in (4), may consist of simple heuristics or different approaches taken from the control theory field.

From a software point of view, monitoring/sensing and acting in a AP is provided by external functions but a AP (adaptation policy) is binded to the sensing/acting libraries it was meant to work with.

Adaptation Manager: AdaM The higher decision making level still maintains the same function it had: it accesses information from the monitors and it decides (in the simplest case through heuristic approach) which AP(s) should be enabled.

In order to do that, during its initialization, AdaM (adaptation manager) has to keep track of which sensing and acting libraries are available in the current system.
This knowledge is the base for the introduction of one of the main aspect of novelty in this work: reinforcement learning.

Communication This modular framework, at run-time, is composed by several processes, one for each AP and one for AdaM (adaptation manager).

Inter-process communication among these processes is implemented through FIFOs, with each process having its own input fifo on which it reads messages sent from other processes.
Typically, each AP will send some messages to AdaM to declare its presence and let it know which sensors and actuator it needs. After having registered the presence of a AP, AdaM will write on the AP’s fifo in order to enable or disable it.

Some other kinds of communications could be in the systems, due to ad-hoc implementation of sensors or actuators. Specifically, in the use of HRM as a sensor, data are passed from kernel- to user-space through shared-memory.

4.3 Artificial Intelligence Approach

The adaptation manager has knowledge and complete access to the interfaces provided by all of the sensing and acting libraries.

Because of this, in the present work we decide to enhance the “adaptivity” of the system allowing it to using policies that are based on sets of sensors and actuators that were not expected to be available.

Not every combination of sensors and actuators might have a specific policy (AP) implemented in the system, different policies could have results that are hard to compare (e.g. if each one is completely failing in reaching one of the objectives), moreover, even if an ad-hoc policy, this could not be the optimal one or not even a suitable one.

At AdaM level, we introduce learning techniques derived from the artificial intelligence field to cope with these situations.

These techniques are reinforcement learning algorithms that are based on Markov decision processes framework.
The motivations for this approach start from the observation of our system: we consider it as the environment of our intelligent agent/controller element. We assume this environment to be contain a single agent and to be fully observable through the sensors available (in this implementation we only rely on HRM but such minimal descriptions have proven to be effective (52)).

However, this environment is an uncertain one because it is unknown “a priori” how actuators affect the environment, so, the transition model is stochastic.

In the most general case, the environment can also be consider sequential, e.g., if the actuator responsible for binding a number of cores to a process only assigns one core at a time (for stability or safety reasons), old decisions will affect the future ones.

We do not know, instead, if the environment is stationary or not, this is also a reason for using reinforcement learning. Active reinforcement learning, with a correct exploration policy, can cope with slowly evolving non-stationary environments.

The type of agent that we want to implement, therefore, is a utility-based learning agent moving in a Markovian environment.

Elements needed in order to do formalize our system into a Markov decision process are:

- **A Set of States**, that can be obtained by Cartesian product and discretization on the domains of each sensor (in our case, intervals of heart rates).

- **A Set of Action**, a (sub-) set of calls to actuator libraries, this is why actuator libraries are implemented by the use of an iterator.
• **A Reward Function**, that can be represented as any distance function of the current state from the state identified by the desired values of each sensor.

Algorithms details are provided in the implementation section.

### 4.4 Novelty of the Proposed Approach

In conclusion, what follows explain why this work should be considered innovative.

• **From an Engineering Point of View**

  1) AdaM and APs is a framework, templates are provided to make easier and faster to create new APs.

  2) AdaM coordinates APs actions considering which sensors/monitors and actuators are available at run time and which ones are required by each AP. This feature was not present in (2).

• **From an Architectural Point of View**

  Using learning, policies are created at run-time without human intervention. This allows to create and integrate new sensors and actuators at any moment, bringing new and unequaled versatility in the system.

• **Performance Considerations**

  The following section presents implementation details of this work, an it is preliminary to the tests and results section, where quantitative analysis of this work is provided.

  The results proposed in (4), where several control strategies are tested in controlling application heartbeat instrumented PARSECs (8), are used as a baseline.
TABLE II
COMPARISON OF PREVIOUS APPROACHES WITH THIS WORK

<table>
<thead>
<tr>
<th></th>
<th>This Research</th>
<th>Previous Researches</th>
<th>Comparison of Control Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Application Heartbeat</td>
<td>Services &amp; Consensus Object</td>
<td></td>
</tr>
<tr>
<td>Presence of Multiple Policies</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Coordination of Policies</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Application Heartbeat</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Heart Rate Monitor</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Observe/Decide Separation</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Decide/Act Separation</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning Capabilities</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible to New Policies</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Extensible to New Sensors</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible to New Actuators</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 5

SYSTEM IMPLEMENTATION

“He is really not so ugly after all, provided, of course, that one shuts one’s eyes, and does not
look at him.”

The Birthday of the Infanta, Oscar Wilde (1892)

This chapter is dedicated to a description and a deep analysis of the implementation of this work.

We present all of crucial elements and passages of the code that has been written in order to perform the tests presented in the following section.

Pseudo-code of some components is provided to allow better understanding.

First, the implementation of communication among modules, which is transversal to the whole system, is explored. Then, implementation of each module/component is introduced: sections two and three are dedicated, respectively, to APs and AdaM, the sections from four to six describe the implementation of sensor and actuator libraries, the last section is dedicated to the description of learning and exploration algorithms.

5.1 Communication

Communication is a relevant issue in the system here implemented, an it is present in multiples forms and at multiple level.
We have stressed how important is modularity in this work. Modularity allows extendability and flexibility that are core properties we want to provide.

However, modularity is introduced by strong separations (e.g. each AP running in its own process, a separated process for AdaM, the functionalities of each actuator encapsulated in a specific library, the implementation of the sensor HRM in kernel-space for performance/security reasons, etcetera) of different kinds that have to be overcome by communication mechanisms of different kinds.

5.1.1 AdaM-APs

AdaM and each AP are represented by processes in a Linux-based operating system. We explained before how these processes implement a two-level decision system. To enforce this structure we need to create some kind of communication between these processes.

Inter-process communication (IPC) in Linux systems can be implemented in several ways, we could use shared memory, mapped memory, pipes, FIFOs or sockets (78).

We choose to use FIFOs (first-in first-out) for two separate reasons. The first one is strictly practical and it explains why we prefer FIFOs over pipes: FIFOs are communication channels with a name in the file-system, so they allow communication between processes that might be unrelated (without parenting relation), as AdaM and APs happen to be.

The second motivation is quite subjective: for seek of simplicity, we prefer to implement each process (AdaM or APs) with a single “listening channel” on which it periodically checks for new messages. This channel is represented by a FIFO opened in “read only” and “non
blocking” modes. It is worth notice the “non blocking” attribute in particular, we do not want AdaM or APs to stall when no message is incoming.

The name of AdaM FIFO is fixed (it actually is “tmpadam_input_fifo”) and it is required to correctly implement APs (that open it in “write only” mode). In practice, developers of new APs do not have to worry about this because this is masked by the functions provided to implement communication.

The name of an AP FIFO is composed of a fixed string and the numerical identifier of the AP. Whenever an AP notifies it existence to AdaM, AdaM open the AP fifo in “write only” mode.

In the end, the communication among AdaM and APs is implemented through a FIFO opened in reading mode by AdaM on which all of the APs write to notify their presence and describe themselves (which sensors and which actuators they need), moreover, each AP has a FIFO opened in reading mode on which AdaM write directives (e.g. enable a given AP on a given “controlled element” that might be a process, a group, the whole system, etcetera).

We can see how the messages incoming in AdaM FIFO might come from several APs at the same moment: it is necessary to implement some synchronization mechanism to avoid inconsistent communication.

We decide to use a mechanism based on messages with header and trailer. Messages are, in fact, treated as sequences of a fixed number of bytes (in this implementation 32 bytes). The “language” of these messages follows some strict rules and it is strongly standardized, so
we used two characters that could not be used inside the message as header and trailer byte, designated to occupy the first and the thirty-second byte of the message.

When a message is processed, if the condition in which the first, and only the first, byte is equal to the header char and the last, and only the last, byte is equal to the trailer char, then the message is discarded.

The content of these messages is described in detail, later, when the general structure of AdaM and APs is discussed.

Communication functions used in AdaM code (underlying functioning is implemented through functions such as, mkfifo(), open(), close(), read(), write() and memcpy() (45) and requires inclusion of sys/types, sys/stat.h, fcntl.h, unistd.h and string.h):

create_and_open_adam_input_channel() This function is used by AdaM to create a FIFO and opening it in non-blocking read-only mode.

read_from_adam_input_channel() This function is used by AdaM to read on its input channel a message of the length of bytes established system-wide.

close_adam_input_channel() This function allows AdaM to close its input channel.

open_ap_channel() This function is used by AdaM to open a FIFO (the one relative to a specific AP) in write-only mode. It takes, as input parameter, an integer representing the identifier of the AP.
write_to_ap_channel()  This function is used by AdaM to write a message to an AP. It takes, 
as input parameters, the ap channel (a file descriptor) and a string (a pointer to a char).

close_ap_channel()  This function is used by AdaM to close the channel to an AP. It takes, 
as input parameter, the ap channel (a file descriptor).

Listing 5.1. Highlights of AdaM communication functions

```c
/* adam_comunication_utils.c */

int create_and_open_adam_input_channel() {
    int mkfifo_err = mkfifo(ADAM_FIFO, 0666);
    // create a new FIFO
    int ip_descriptor = open(ADAM_FIFO, O_RDONLY|O_NONBLOCK);
    // open the FIFO in read-only, non-blocking mode
    [...]
    // produce error code
}

int open_ap_channel(int ap_identifier) {
    int temp = ap_identifier;
    char* base = "/tmp/ap";
    char* path;
    // create a string pointed by path containing the expected name of the AP FIFO
    int mkfifo_tmp = mkfifo(path, 0666);  // create a new FIFO
    int ap_descriptor = open(path, O_WRONLY);  // open the FIFO
    [...]
    // produce return value
}
```

Communication functions used in an AP code:
**open_adam_channel()** This function is used by a AdaM to open the FIFO relative to AdaM in write-only mode. It takes, as input parameter, an integer representing the identifier of the AP.

**write_to_adam_channel()** This function is used by a AdaM to write a message to a AP (see below more about these messages). It takes, as input parameter, a string (a pointer to a char).

**close_adam_channel()** This function is used by an AP to close the channel to AdaM.

**create_and_open_ap_input_channel()** This function is used by an AP to create a FIFO and opening it in non-blocking read-only mode.

**read_from_ap_input_channel()** This function is used by an AP to read on its input channel a message of the length of bytes established system-wide.

**close_ap_input_channel()** This function allows an AP to close its input channel.

---

**Listing 5.2. Highlights of AP communication functions**

```plaintext
1 /* ap.comunication_utils.c */
2 [...]  
3 int write_to_adam_channel(char* message) {
4     int ret = write(ap_type.adam_descriptor, message, MESSAGE_LENGTH);
5     // write messages of fixed length to the ADAM FIFO
6     [...] // produce error code
7 }    
8 [...] 
9 [...] 
```
As we said while proposing this approach, the sensor/monitor of choice in this work is heart rate monitor (HRM).

HRM implementation, even if not directly related to this work, is briefly touched later. For what matters in communication, it is just worth notice that HRM is implemented in kernel-space.

**AdaM/APs-HRM**

Figure 9. Communication flows
Decision mechanisms are implemented in AdaM as well as in each AP and these mechanisms rely on “percepts” of the system that are provided by HRM. Consequently, we need a way to interface the user-space processes in which AdaM and APs run with HRM kernel-module.

This part of communication is realized through shared-memory. A wrapper library (a library exposing a simple interface of a complex software element) has been implemented to attach HRM in any user-space process and recall several statistics, such as actual heart rates, target heart rates, relations between “groups” and “threads” (5), etcetera.

This library goes under the name of “libhrm” and it is static library. Static libraries require more memory and produce bigger executable files but the provide faster execution and a different copy loaded to memory for each process using them.

To use it, it is sufficient to include the “-lhrm” flag in compiling our executable files.

Specific functions are used to attach and detach the segment of memory relative to HRM.

**hrm.t** This is the type used to keep the information about the monitor when we want to use HRM in user-space applications.

**hrm.t *hrm_attach(int gid, bool consumer)** This function allows to attach the shared memory containing the information collected by HRM about a “group”. It takes, as input parameters, the identifier of a group and a boolean saying whether or not the attaching application will emit heartbeats. It return a pointer to the data type just introduced.

**int hrm_detach(hrm.t *monitor)** This function allows to detach the shared memory containing the information collected by HRM about a “group”. It takes, as input parameters,
the hrm data type whose pointer was returned by the relative attach. It return an error
code.

\texttt{uint32_t hrm_get_global_heart_rate(const hrm_t *monitor)} This function returns the
heart rate relative to the whole life of a “group”. It takes, as input parameters, the
hrm data type whose pointer was returned by the relative attach.

\texttt{uint32_t hrm_get_window_heart_rate(const hrm_t *monitor)} This function returns the
heart rate relative to the most recent window of time of a “group”. It takes, as input
parameters, the hrm data type whose pointer was returned by the relative attach.

\texttt{uint32_t hrm_get_min_heart_rate(const hrm_t *monitor)} This function returns the min-
imum desired heart rate relative to a “group”. It takes, as input parameters, the hrm
data type whose pointer was returned by the relative attach.

\texttt{uint32_t hrm_get_max_heart_rate(const hrm_t *monitor)} This function returns the max-
imum desired heart rate relative to a “group”. It takes, as input parameters, the hrm
data type whose pointer was returned by the relative attach.

\texttt{size_t hrm_get_window_size(const hrm_t *monitor)} This function returns the window
size (time interval considered by the “hrm_get_window_heart_rate” function above) rel-
ative to a “group”. It takes, as input parameters, the hrm data type whose pointer was
returned by the relative attach.

\texttt{int64_t hrm_get_timer_period(const hrm_t *monitor)} This function returns the timer
period (it forces a frequency for computation of some metrics, it is not used in this
implementation). It takes, as input parameters, the hrm data type whose pointer was returned by the relative attach.

**pid_t **hrm_get_tids(const hrm_t **monitor)** This function returns a zero-terminated array of integers containing the thread identifiers of all the threads relative to a “group”. It takes, as input parameters, the hrm data type whose pointer was returned by the relative attach.

```
Listing 5.3. Sample code of a process using the HRM wrapper
/* hrm_wrapper_example.c */

#include <hrm.h>

int main(int argc, char *argv[]) {

    hrm_t *monitor = NULL;
    [...]
    monitor = hrm_attach(GROUP_ID, true);
    // attach the monitor to a group as a consumer
    while (1) {
        pid_t *tids = hrm_get_tids(monitor);
        // obtain an array with the group TIDs
        uint32_t hr_w hrm_get_window_heart_rate(monitor);
        // obtain the window heart rate of the group
        [...]
    }
    return 0;
}
5.2 **Structure of a Generic AP**

Each AP is a process responsible for monitoring the system it is running in and acting on it. Generally speaking, an AP can use any number of sensors to monitor the system and use any number of actuators to modify/influence the system behaviour.

Furthermore, the area of the system on which an AP performs its task is limited by a list of identifiers. For example, in the implementation of an AP that monitors through HRM a set of applications and uses the “taskset” implementation of a core-allocating actuator, these identifiers will be the “group identifier” of the monitored applications.

It is not necessary to impose strict rules on what these identifiers are or they represent, however, it is mandatory in the “decide” phase of an AP, its policy, to be able to disambiguate consistently these values (this burden is left to the AP developer). Moreover, while implementing new sensors and actuators, it should be a good custom to introduce the capability to cope with unexpected identifiers through special output values.

In the case an AP is responsible for “system-wide” monitoring and acting, we can use a special identifier for the system (this is not part of this implementation but we suggest to use the root process identifier).

The general functioning of an AP is relatively simple, as soon as it is create, an AP notifies its presence to AdaM and registers to it providing a brief description of itself (sensors and actuators used). Then, the AP periodically checks its input FIFO for new messages from AdaM.
The content of these messages are the identifiers on which the AP has to perform its decision making.

APs differ from one another by the decision making they use and the set of sensors/actuators exploited. In this implementation, all of the sensors and all of the actuators have publicly known identifiers.

Functions provided to implement new APs (the identifier of an AP can be modified through a DEFINE in its header file):

- **ap_t** It is the data type used to group all the information used by the following service functions.
- **initialize_ap()** This function initializes the data type above with the basic working conditions of the AP (identifier, no monitored identifiers, etcetera).
- **start_ap_registration()** After the creation of the communication channel with AdaM, this function is used to notify it with the AP identifier.
- **required_sensor_registration()** Then, this function is used to notify AdaM with each of the sensors required by the AP.
- **required_actuator_registration()** This function is used to notify AdaM with each of the actuators required by the AP.
- **complete_ap_registration()** After notifying the sensors and the actuators required, the AP completes its registration to AdaM.
- **check_message_integrity()** This function checks that a message received on the input FIFO has not been corrupted. It is usually trivial in the AP situation: each AP is supposed to
receive messages only by the only existing AdaM in the system, but a similar function is
used AdaM itself.

disambiguate_message() This function reads the new message and elaborate its content,
i.e., it follows the directions of AdaM in adding or removing identifiers from the list of
controlled identifiers of an AP.

next_identifier() This function is simply an iterator over the identifiers controlled by the AP.

get_percepts_for_identifier() This function is used to obtain the current percepts relative to
a controlled identifier by the AP. It returns a list of values, one for each sensor. Minimal
intervention of the AP developer might be required here (sensor libraries, as well as
actuator libraries have standardized calls, however if we were using 3rd party or later
developed libraries, some code adjustments could be necessary).

decide() This function take as input the output list of the previous one and it performs the AP
specific decision making on it. Its implementation is responsibility of an AP developer. In
this work, several APs have been created by reformulating some of the decision techniques
seen in (4). It returns a list of values, one for each actuator.

act() This function take as input the output list of the previous one and it applies it through the
relative actuator libraries. This is actually optional, the AP developer could choose to use
functions exported by the actuator libraries inside the decide phase, but recommended.

Listing 5.4. Pseudo-code of an AP

/* ap_template.c */
# include "ap_template.h"

[...]
ap_t ap; // global variable

int* decide ( percept_node* list_of_percepts ) {
    // several possible policy implementation
}

int main ( int argc , char* argv[] ) {

    int err = initialize_ap ( &ap );
    err = start_ap_registration ( &ap )
    [...]
    required_sensor_registration ( &ap , SENSOR_1_IDENTIFIER )
    required_actuator_registration ( &ap , ACTUATOR_1_IDENTIFIER )
    required_actuator_registration ( &ap , ACTUATOR_1_IDENTIFIER )
    err = complete_ap_registration ( &ap )
    [...]

    while ( /* LOOP*/ ) {
        // read input from adam
        err = check_message_integrity ( message );
        [...]
        disambiguate_message ( &ap , message ); // use the information in the message to update ap

    while ( next_identifier ( &ap ) != NULL ) {
        identifier_t current = next_identifier ( &ap );
        percept_node* percepts = get_percepts_for_identifier ( current );
        int* actions = decide ( percepts );
        err = act ( actions );
    }

    // close/clean all
    return 0;
}
AdaM is a process keeping a lot of knowledge about the system and responsible for several high-level decisions.

It is supposed to be the first process to be run in the system (before the APs, actually if AdaM does not exists, APs will stall trying to open AdaM FIFO to register on it).

AdaM is able to run on several system configuration. By system configurations we mean context in which different sensors and actuators are available and, consequently, different sensor and actuator libraries have to be included.

In the implemented template, using simple code conventions, all the known sensor and actuator libraries are present and their inclusion can be modified through some DEFINEs. These DEFINEs are also used, at run-time, as fixed parameters when it comes to decide whether an AP should be allowed to run in the system or not (i.e. it the sensors and the actuators required by the AP are available).

So, AdaM has static complete knowledge of the sensors and actuators of the system and dynamic knowledge of the registered APs. Indirectly, through the sensors, AdaM is able to obtain and keep dynamic knowledge of all the identifiers of the controlled elements in the system.

The elaboration of complex decision policies to be integrated in AdaM is not part of this work, heuristics and a machine learning approach were proposed in (2). Instead, in this work, we add a new choice for AdaM beyond the APs.
We add to AdaM the capability of learning on-line a new policy using all of the sensors and all of the actuators available at that time.

A schematic representation of how AdaM works is given in the pseudo-code in listing 5.5.

```
Listing 5.5. Pseudo-code of AdaM

/* adam.c */
#include "adam.h"

int main (int argc, char* argv[]) {
    int err = initialize_adam();
    [...] 
    while (/*LOOP*/) {
        // read input messages
        err = check_message_integrity(message);
        [...] 
        disambiguate_message(message); // use the information in the message to update knowledge about aps
        while (next_sensor(&adam) != NULL) {
            identifier_t current = next_sensor(&adam);
            // read info from each sensor
        } 
        // decide whether use identifiers to ap mapping or reinforcement learning
        while (next_ap(&adam) != NULL) {
            identifier_t current = next_ap(&adam);
            // write to each sensor
        } 
        if (/*reinforcement learning*/) {
            // run learning algorithm and act
        }
        [...] 
    // close/clean all
    return 0;
}
```
5.4 **Sensor: HRM**

Detail about the implementation of the sensor HRM can be found in (5) and (77), here we only report some sentences from the first of the two sources that allow to better understand its functioning.

“A task, which is either a process or a thread, represents a unit of execution. An application is an ensemble of tasks pursuing a set of objectives (e.g., encoding a video stream); hence, an application may be single-threaded, multi-threaded, multi-processed, or any feasible combination of them. A heartbeat is a signal emitted by any of the applications tasks and stands for an execution progress.”

“A group is a subset of applications tasks pursuing a common objective [...] and it is identified by a unique Group IDentifier (GID).”

“Each group is characterized by its counter, which records heartbeats, statistics, which contain the heart rate (i.e., the frequency at which tasks emit heartbeats), and target, which is expressed as a heart rate range.”

“HRM is partitioned between user- and kernel-space. [...] The kernel-space partition consists of an API that mimics libhrm, which allows building kernelspace adaptation policies, and of the core of the performance monitor extending Linux in few key points. HRM introduces include/linux/hrm.h and kernel/hrm.c, and modifies include/linux/sched.h and fs/proc/base.c.”
5.5 Actuator: Core Allocator with “Taskset” Command

We identify and use several ways to affect system’s behaviour. These are encoded into actuator libraries. The first one we propose is derived from an idea already present in early works about application heartbeat, such as, for example (3).

We use the Linux “taskset” command to influence the scheduling of processes/threads over cores of a multi-core processor.

“Taskset is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. CPU affinity is a scheduler property that bonds a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. [...]”

The CPU affinity is represented as a bit-mask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU.” (45)
The taskset core allocator library is implemented as a shared library (only one copy is kept in memory and we can have lighter APs executable).

While in the previous implementations of system similar to this one, such as (9) and (4), the taskset command was used as an abstract actuator without a precise definition and encapsulation of it, we use a library to define an actuator to bound it to the idea of a “set of actions” (a concept often seen in artificial intelligence). This logical restriction allows to exploit the reinforcement learning algorithms presented later.

The set of actions relative to a command with “n” input parameters is represented by the Cartesian product of the domains of each parameter. Essentially, the library provides different implementations of the same actuator considering different sub-sets of this product.

In fact, the set of actions could be represented by all the possible masks applied to every existing process (if we are making decisions system-wide) but also the choice of considering only some masks for a fixed process could be rational (if we are making decisions looking at one process at a time).

The library contains several functions, relative to different action-sets. However, during the test phase of this work we mainly focus on two sets: the set of all the possible consecutive masks for a given process and the set of masks that differ by one in cardinality w.r.t. the current mask for a given process.

Some of the functions provided are (the number of cores is fixed at compile-time by a DEFINE):
**cores_allocator_t** This is the data type used to keep all the information needed by the following service function, such as the current mask of a given process identifier.

**ca_init()** This function initializes the data type above with its default values.

**ca_add_identifier()** This function adds an identifier of processes to which apply CPU affinities to the core allocator data type.

**ca_remove_identifier()** This function removes an identifier of processes to which apply CPU affinities to the core allocator data type.

**ca_get_cardinality_processwide_bystep()** This function returns the number of available actions when we modify CPU affinity process-wide and by a single step (i.e. three actions, “assign one more core”, “assign one core less”, “assign the current number of cores”).

**ca_do_nth_action_processwide_bystep()** This function performs one of the actions mentioned above. It takes as input an index representing the action.

**ca_get_cardinality_processwide_byrange()** This function returns the number of available actions when we modify CPU affinity process-wide and choosing a range of cores (e.g. on a four cores processor, we will have four actions).

**ca_do_nth_action_processwide_byrange()** This function performs one of the actions mentioned above. It takes as input an index representing the action.

---

Listing 5.6. Highlights of CA library code

```c
/* core_allocator_library.c */
```
5.6 **Actuator: Frequency Scaler**

The frequency change is supported by the “cpufrequtils” package, a set of userspace utilities designed to assist with CPU frequency scaling.

Through the frequency scaler library, we first set the governor to “userspace” (i.e. we make the cores frequency manually tunable), then we act on the frequency values through “scaling_setspeed”.

Just like the core allocator library, the frequency scaler library is implemented as a shared library and to be linked to an AP.

Considerations, similar to those we made for the core allocator implementation, can be made about “action sets” for the frequency scaler.
The situation, in this case, is simpler for two reasons: the distinction about process-wide and system-wide decision does not exist anymore (independently from the scope of the decision, the action will have the entire system as its scope); moreover in the tested architecture we only can choose a frequency value for all of the cores at a time.

So, in this implementation we provide functions to use two different set of actions for the frequency scaling: changing the frequency one step at a time or setting it to any of the possible steps.

It is worth notice that the two implementation allow to reach the same configurations but with different sequences of actions.

**frequency_scaler_t** This is the data type used to keep all the information needed by the following service function, such as the current frequency.

**fs_init()** This function initializes the data type above with its default values.

**fs_get_cardinality_systemwide_bystep()** This function returns the number of available actions when we modify the frequency by a single step (i.e. three actions, “step up”, “step down”, “stay where you are”).

**fs_do_nth_action_systemwide_bystep()** This function performs one of the actions mentioned above. It takes as input an index representing the action.

**fs_get_cardinality_systemwide_byvalue()** This function returns the number of available actions when we modify the frequency choosing which step we want to obtain (in this case we will have as many actions as steps).
**fs_do_nth_action_systemwide_byvalue()** This function performs one of the actions mentioned above. It takes as input an index representing the action.

```c
/* core_allocator_library.c */

[...]

int ca_do_nth_action_processwide_bystep(ca_t* ca, identifier_t process, int n) {

[...]

memcpy(&numcores, ca->cores_allocated_to_pid + (pid_index * sizeof(int)),
    sizeof(int));

// obtain the num of cores in the current process affinity mask
int choose = n%3;
numcores = //update numcores
char command[256];
// issue the command
sprintf(command,"tasksset-pc%d-%d", numcores-1, process);
int retval_tasksset = system(command);
if (retval_tasksset==1) {perror("tasksset_error"); return 1;}
[...]

[...]
```

### 5.7 Reinforcement Learning

As we explained before, the AdaM element keeps a broad knowledge of the system in execution. AdaM knows about all the sensors available to collect information and all the actuators libraries that can be used to tune the system performance.
In this implementation AdaM can use the information coming from the sensors to decide how to assign identifiers to be controlled among APs (or vice versa, which AP to enable on which identifier) but it can also do more.

AdaM can choose to take over complete control over the system applying a new system-wide policy. However, this policy cannot be hard-coded in AdaM because it could be running in conditions in which only some sensors/actuators are available, or perhaps, new sensors/actuators have been just introduced.

This is why we introduce in AdaM the ability to learn a new policy through the use of machine learning. In order to do that, we formalize the problem as a Markov decision process. For it, to use reinforcement learning algorithms able to learn the optimal policy, we have to specify three components:

- A set of states.
  \[ S = ([x_1, x_2, \ldots, x_n] \mid x_1 \in \mathbb{R}^+, \text{etc.}) \]
  In the actual tests \( n = 1 \), and this single values represents which interval (out of a discrete number of intervals in which the domain is divided) the heart rate of the controlled application is.

- A set of actions.
  \[ A = ([a_1, a_2, \ldots, a_n]) \]
  It depends on whether or not we are using both core allocation and frequency scaling and through which implementation. Functions “get_cardinality” comes handy in this case.
One of the reasons why we use reinforcement learning is its ability to learn (utilities or transitions) without the need of prior knowledge about actions semantic. Therefore, as long as the possibility to iterate over a finite number of actions is available, we can combine the calls exported by the actuators libraries in several different way. In our tests we use three different set of actions with different cardinalities.

- A reward function to be applied to states. In general the reward could be proportional to the proximity to a desired point in a k-dimensional space (where k is the number of sensors).

\[
R([x_1, x_2, \ldots, x_K]) = \frac{1}{K} \sum_k \left( \frac{const}{1 + \left(\text{current}_\text{value}(k) - \text{desired}_\text{value}(k)\right)^2} \right)
\]

As we said, in our test, the state is one-dimensional so the current\_value will be the measured heart rate and the desired\_value is actually a range of heart rates. Because of this, the reward function is a piecewise-defined function, positive in the desired range, slightly negative in the adjacent ranges and so on so forth.

Below we provide, through pseudo-code, description of the different learning algorithms used and of the different exploration strategies to perform learning (in fact, we are considering active learning performed on-line).

**Adaptive Dynamic Programming** An ADP agent uses counters of the actions taken in each state and of the states it ends in to learn a transition model and then solve the
associated MDP, computing the utility of each state. From this point, the optimal policy is derived straightforward by using these utilities and the learnt transition function.

Listing 5.8. Pseudo-code of an ADP agent

```c
/* adp.c */

typedef struct {
  // fields
} state;
typedef enum { /* actions */ } action;

int observed[NUM_STATES];
double Utilities[NUM_STATES];
double Rewards[NUM_STATES];
int State_Action_Pair_Attempts[NUM_STATES][NUM_ACTIONS];
int State_Action_Pair_Results[NUM_STATES][NUM_ACTIONS][NUM_STATES];
double Learnt_Transition_Function[NUM_STATES][NUM_ACTIONS][NUM_STATES];

action adp_learning_agent(state current_state, double current_reward) {
  // initialization of Reward[], Utilities[] and observed[]
  State_Action_Pair_Attempts[last_state][last_action]++;
  State_Action_Pair_Results[last_state][last_action][current_state]++;
  for (/* every state */) {
    // update Learnt_Transition_Function
  }
  compute_utilities_and_solve_current_MDP();
  last_state = current_state;
  last_action = exploration_strategy(current_state);
  [...] 
  return last_action;
}
```
Q-Learning A Q-learning agent uses an update-rule to learn the utilities of action-state pairs, consequently, in each state, the action associated with the highest action-state utility is the optimal choice.

Listing 5.9. Pseudo-code of a Q-learning agent

```c
/* q-learning.c */
typedef struct {
    // fields
} state;
typedef enum {/* actions*/} action;
double Q_values[NUM_STATES][NUM_ACTIONS];
int State_Action_Pair_Attempts[NUM_STATES][NUM_ACTIONS];

action q_learning_agent(state current_state, double current_reward) {
    // initialization of Q_values/
    State_Action_Pair_Attempts[last_state][last_action]++;
    Q_values[last_state][last_action] =
        Q_values[last_state][last_action] + alpha * (last_reward + (gamma *
            max_Q_for_state_over_actions(current_state))
        - Q_values[last_state][last_action]);
    last_state = current_state;
    last_reward = current_reward;
    last_action = exploration_strategy(current_state);
    [...] 
    return last_action;
}
```

SARSA A state-action-reward-state-action agent is very similar to a Q-learning agent but it uses a different update-rule.
Every active learning agent requires an exploration strategy. Exploration strategies are the way in which the agent decides whether to exploit its knowledge of the environment/system or to continue (and how) exploring it to better understand it.
Every exploration strategy is a way to cope with the unsolved (for multi-states systems) problem of exploration-exploitation trade-off. Here we presents the strategies implemented and used in this work.

**Greedy Exploration** A greedy exploration strategy forces the agent to random (uniformly distributed) actions until the learning algorithm does not reach convergence.

```
/* greedy_exploration.c */

action GREEDY_EXPLORATION(current_state) {
  IF (learning_algorithm_reached_convergence == TRUE) 
    RETURN optimal_policy(current_state);
  ELSE 
    RETURN random_action_from(actions_available_in(current_state));
}
```

**Counter-Based Exploration** A counter-based exploration strategy forces the agent to take random actions as long as it has not tried some state-action pairs for a sufficient number of times.

```
/* counter_based_exploration.c */

action COUNTER_BASED_EXPLORATION(current_state) {
  IF (ad_hoc_counter(current_state) > THRESHOLD) 
    temp_action optimal_policy(current_state);
  update_counter(current_state, temp_action);
  RETURN temp_action;
```
Recency-Based Exploration A recency-based exploration is similar to the previous one but in this case state-action pairs counters decrease over time, making necessary to re-try actions that have not been taken for too long.

Listing 5.13. pseudo code of learning agent

```c
/* recency_based_exploration.c */

action RECENTY_BASED_EXPLORATION( current_state ) {
    update_counters_of_all_actions;
    IF ( ad_hoc_counter( current_state ) > THRESHOLD)
        temp_action = optimal_policy( current_state );
    update_counter( current_state , temp_action );
    RETURN temp_action;
ELSE
    temp_action =
        ad_hoc_prob_distribution_over( actions_available_in( current_state ));
    update_counter( current_state , temp_action );
    RETURN temp_action;
}
```

As we saw in the chapter dedicated to the theoretical background of this work, the first exploration strategy described falls in the undirected exploration techniques family while the other
two are directed (counter-based) exploration techniques. In (35) and (34), Sebastian Thrun de-
scribes other more complex strategies. Even if interesting and promising, the implementation
of such techniques is beyond the scope of this work.
CHAPTER 6

TEST SET UP AND RESULTS

“Nothing comes out of nothing.”
Principia Philosophiae, René Descartes (1644)

This chapter contains the description of the tests we performed in order to evaluate the system whose implementation has just been discussed.

These quantitative tests focus on the newly introduced learning ability of the system.

The first two sections of this chapter describe the hardware on which we performed our tests and the suite of benchmarks (intended for research) used for them. The third section describes the setting up of these tests and how used parameters were chosen. Then, in the fourth section, for each tested benchmark, extensive presentation of its results is given through graphs and tables.

6.1 Testing Environment

These are the specification of the machine we used for our tests:

**Processor**: Intel Core-i7 870, Quad-Core

**Frequency**: Maximum 2.93 GHz (Minimum 1.20 GHz)

**Cache**: 8 MB of shared LLC (Last-Level Cache) L3

**Memory**: 4 GB DDR-1333
6.2 PARSEC Benchmark Suite

To perform the tests in this section we used the PARSEC benchmark suite developed from Princeton University (8).

Features of this suite:

Multi-threaded, PARSEC is a parallel benchmark suite.

Workloads, PARSEC includes workloads meant to be the most relevant in the future.

Diverse Application Domains are explored.

Not High Performance Computing-Focused because HPC applications are only a small sub-set of the whole application space.

Intended for Research

Programs The complete suite is composed by 13 programs (8), we use most of them:

- blackscholes - Option pricing with Black-Scholes Partial Differential Equation (PDE)
- bodytrack - Body tracking of a person
- canneal - Simulated cache-aware annealing to optimize routing cost of a chip design
- dedup - Next-generation compression with data deduplication
- ferret - Content similarity search server
- fluidanimate - Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
- raytrace - Real-time ray-tracing
**swaptions** - Pricing of a portfolio of swaptions

**x264** - H.264 video encoding

All of the used PARSEC programs have been properly instrumented with HRM calls. In bodytrack, facesim, fluidanimate, raytrace and x264, each heart beat represents a frame, in blackscholes an HB is an option, in canneal an HB is a swap, in dedup an HB is a chunk, in ferret an HB is a query, in streamcluster an HB is a point and in swaptions an HB is an experiment (77).

### 6.3 Tests Description

In the following we presents the results of our tests. We tested the performance of six different rational agents differing for the learning algorithm implemented and the set of possible actions available.

In this part we chose to focus more on using different actions for two main reasons:

1. To prove that this approach is flexible and able to integrate seamlessly different “actuators”.

2. Because, in preliminary experimental results, tests on different actions sets created the case for more interesting observations than using different exploration policies.

Moreover, being the SARSA algorithm (already discussed in (4)) very similar to the Q-learning approach but more sensitive to the exploration strategy used, because of its on-policy nature, we only focus on the adaptive linear programming algorithm (presented in the theoretical background chapter) and on Q-learning.
The six agents (three ADP agents and three Q-learning agents) uses three different sets of actions extracted from the actuators used to modify the processes affinity masks and the CPU frequency.

1. First set of actions (five actions): assign one core more to process affinity mask, assign one core less to process affinity mask, increase the CPU frequency by one step, decrease the frequency by one step, do nothing.

2. Second set of actions (ten actions): assign a process affinity mask with one core, ..., assign a process affinity mask with four cores, set the CPU frequency to the first step, ... (the second, sixth and tenth step), set the CPU frequency to the fourteenth step.

3. Third set of actions (nine actions): the actions in the first set and the four combined (affinity mask an frequency modification) actions.

All of the agents are tested on ten consecutive runs of nine different PARSEC benchmarks.

Having previous knowledge about statistics of HRM instrumented PARSECs on the tested machine (77), we choose to assign a positive reward (to attract our agents) for a “heart rate window” (thick blue lines), centered on 50% of the “maximum average known heart rate” of that PARSEC and with width equal to 20% of the “maximum average known heart rate” of that PARSEC. The ranges from 20 to 40% and from 60 to 80% are “neutral zones” (thin blue lines), where the agent could settle if it is not able to get to the best window, or if it fails to explore the environment enough. Every other heart rate value has great negative reward.
6.4 Tests Commentary

In the following pages, we provide the graphics and extended commentary of the executions of all of the nine PARSECS with all of the six agents. Moreover, an overview can be obtained by looking at a summary table containing the average heart rate of each agent and each PARSEC.

This table contains also an error measure that is defined as the squared root of the mean squared error (MSE), in this case, the distance from the highest rewarded heart rate range, normalized by the heart rate of the application with full resources available (the “maximum average known heart rate”). This measure is introduced to allow a fair comparison of different agents on different PARSECs.

6.4.1 Blackscholes

In Blackscholes (Figure 11 to Figure 16) we can see that learning happens quickly for all the agents. After the forced exploration at the beginning, all of the agents start applying a policy that does not change much in the following executions.

All of the agents decide to assign the application to two cores for most of the time and this allow them to keep the heart rate in the highest rewarded range.

Exceptions are the ADP agent with the first set of actions that apparently fails at fully learning during exploration and settle for “acceptable” range, assigning the application to only one core.

Interesting is the solution found by the Q-learning agent with the third set of actions that learns to assign the application to four cores and uses the frequency setting to keep the application heart rate at the lower border of the highest reward range.
Figure 11. Blackscholes, ADP agent, first set of actions, avg. hr 5490257, % error 0.23

Figure 12. Blackscholes, ADP agent, second set of actions, avg. hr 10698401, % error 0.16

Figure 13. Blackscholes, ADP agent, third set of actions, avg. hr 9648170, % error 0.11

Figure 14. Blackscholes, Q-learning agent, first set of actions, avg. hr 9782523, % error 0.14

Figure 15. Blackscholes, Q-learning agent, second set of actions, avg. hr 10614396, % err 0.12

Figure 16. Blackscholes, Q-learning agent, third set of actions, avg. hr 7960482, % error 0.12
6.4.2 Bodytrack

The Bodytrack benchmark results (Figure 17 to Figure 22) are influenced by the fact that, on the tested machine, this application can achieve heart rates of only few (less than five) heartbeats per second. HRM allows only for integer heart rates values so, the only possible value for the highest reward range is three heartbeats per second.

All of the agents learns quickly but none of them finds a way to keep the heart rate stable to the value of three (notice that a such a configuration of the system might not exist).

All of the agents, except for the ADP agents with the second actions set, try to control the application making it oscillating around three, between the “acceptable” values of two and four heartbeats per second.

6.4.3 Canneal

Canneal (Figure 23 to Figure 28) can be kept in the highest rewarded heart rate range with multiple configurations (e.g. two or three cores with different frequencies). All of the agents use one or more of these configurations and are able to correctly control the application.

It is only worth notice that, at the beginning of each execution, the PARSEC presents some sort of uncontrollable spike related to the way heartbeats have been placed in its code.
Figure 17. Bodytrack, ADP agent, first set of actions, avg. hr 1.7, % error 0.21

Figure 18. Bodytrack, ADP agent, second set of actions, avg. hr 1.8, % error 0.18

Figure 19. Bodytrack, ADP agent, third set of actions, avg. hr 1.2, % error 0.22

Figure 20. Bodytrack, Q-learning agent, first set of actions, avg. hr 1.8, % error 0.18

Figure 21. Bodytrack, Q-learning agent, second set of actions, avg. hr 2.2, % error 0.19

Figure 22. Bodytrack, Q-learning agent, third set of actions, avg. hr 1.8, % error 0.20
Figure 23. Canneal, ADP agent, first set of actions, avg. hr 979257, % error 0.11

Figure 24. Canneal, ADP agent, second set of actions, avg. hr 1125487, % error 0.11

Figure 25. Canneal, ADP agent, third set of actions, avg. hr 1075179, % error 0.11

Figure 26. Canneal, Q-learning agent, first set of actions, avg. hr 1121591, % error 0.12

Figure 27. Canneal, Q-learning agent, second set of actions, avg. hr 1130942, % error 0.12

Figure 28. Canneal, Q-learning agent, third set of actions, avg. hr 863815, % error 0.10
6.4.4 Dedup

The Dedup PARSEC (Figure 29 to Figure 34) appears to be almost uncontrollable (with these goals) by these agents on this machine. All of the ADP agents learn quickly but they are only able to keep the application in the “acceptable” ranges (the upper one with the first two actions sets and the lower one with the third actions set) but basically never reach the highest rewarded heart rate range.

The Q-learning agents seems to perform in general even worse, the Q-learning agent with the second actions set learns slowly and only an “acceptable” policy. However, the Q-learning agent with the first actions set seems to find a way to correctly control the application just after the initial exploration, but then it lose this knowledge and it does not seem able to recover. The same happens the the Q-learning agent with the third actions set. This agent, though, seems to recover during the last execution.

We can argue that Q-learning, even if converging very slowly, would find a way to control this PARSEC.

6.4.5 Ferret

Despite showing a quite intrinsic variance, probably due to the specific HRM instrumentation, this PARSEC (Figure 35 to Figure 40) seems to be controlled well enough (when not in the highest rewarded range, it is kept in the “acceptable” range).

Moreover, two interesting patterns emerge. The agents with the second set of actions clearly are outperformed by the other agents. The agents using the Q-learning algorithm get to a stable control later than the ADP agents (they learn slower).
Figure 29. Dedup, ADP agent, first set of actions, avg. hr 5982, % error 0.49

Figure 30. Dedup, ADP agent, second set of actions, avg. hr 6386, % error 0.57

Figure 31. Dedup, ADP agent, third set of actions, avg. hr 2874, % error 0.34

Figure 32. Dedup, Q-learning agent, first set of actions, avg. hr 5956, % error 0.64

Figure 33. Dedup, Q-learning agent, second set of actions, avg. hr 7663, % error 1.02

Figure 34. Dedup, Q-learning agent, third set of actions, avg. hr 4938, % error 0.47
Figure 35. Ferret, ADP agent, first set of actions, avg. hr 12.0, % error 0.18

Figure 36. Ferret, ADP agent, second set of actions, avg. hr 15.0, % error 0.25

Figure 37. Ferret, ADP agent, third set of actions, avg. hr 13.4, % error 0.15

Figure 38. Ferret, Q-learning agent, first set of actions, avg. hr 13.3, % error 0.17

Figure 39. Ferret, Q-learning agent, second set of actions, avg. hr 15.5, % error 0.28

Figure 40. Ferret, Q-learning agent, third set of actions, avg. hr 14.6, % error 0.23
6.4.6 Fluidanimate

For the Fluidanimate PARSEC (Figure 41 to Figure 46), we can distinguish two kinds of behaviour by the agents. The first one is the behaviour showed by the ADP agent with the first set actions and the Q-learning agent with the third set of action. Even if they have very different learning time (the ADP agent learns is quicker), they both settle for the lower “acceptable” range and show a minimal oscillating behaviour.

The other four agents, on the other hand, have an oscillating behaviour but they are able to do that around a heart rate value inside the highest rewarded heart rate range and close to its upper border.

6.4.7 Raytrace

All of the agent manage to learn quickly and to keep the Raytrace benchmark (Figure 47 to Figure 52) heart rate inside the “acceptable” ranges. However, most of them do not keep the value inside the highest rewarded range in a consistent way and a strong oscillating behaviour is often present.

Only the Q-learning agent with the second actions set is able to keep the application heart rate inside the best range with only rare oscillation in the lower “acceptable” agent.
Figure 41. Fluidanimate, ADP agent, first set of actions, avg. hr 7.2, % error 0.14

Figure 42. Fluidanimate, ADP agent, second set of actions, avg. hr 11.5, % error 0.16

Figure 43. Fluidanimate, ADP agent, third set of actions, avg. hr 10.5, % error 0.15

Figure 44. Fluidanimate, Q-learning agent, first set of actions, avg. hr 11.0, % error 0.15

Figure 45. Fluidanimate, Q-learning agent, second set of actions, avg. hr 10.9, % error 0.15

Figure 46. Fluidanimate, Q-learning agent, third set of actions, avg. hr 9.4, % error 0.16
Figure 47. Raytrace, ADP agent, first set of actions, avg. hr 6.8, % error 0.14

Figure 48. Raytrace, ADP agent, second set of actions, avg. hr 7.2, % error 0.17

Figure 49. Raytrace, ADP agent, third set of actions, avg. hr 6.7, % error 0.17

Figure 50. Raytrace, Q-learning agent, first set of actions, avg. hr 6.1, % error 0.19

Figure 51. Raytrace, Q-learning agent, second set of actions, avg. hr 7.0, % error 0.14

Figure 52. Raytrace, Q-learning agent, third set of actions, avg. hr 5.8, % error 0.19
6.4.8 Swaptions

The Swaptions PARSEC (Figure 53 to Figure 58) appears to be one of the easiest to control, all of the agents learn quickly during the mandatory exploration phase and, after that, are able to keep the application heart rate inside the highest rewarded range, assigning the application to two cores.

Only the Q-learning agent with the first set of actions, though starting the post-exploration phase well, it then drops to the lower “acceptable” range, assigning the application to only one core, and it settles for it, without being able to recover (this might actually be fixed by a more complex exploration policy).

6.4.9 x264

The x264 PARSEC (Figure 59 to Figure 64) does not appear to be easy to control. However, in general, ADP agents are able to understand and learn a correct policy better than Q-learning agents.

All of the ADP agents are able to keep the application heart rate almost always inside the “acceptable” range and very often inside the highest rewarded one. Among the ADP agents, the one using the second set of actions (that in other cases, e.g., the Ferret PARSEC, was the least useful one) appears to be the best one.
Figure 53. Swaptions, ADP agent, first set of actions, avg. hr 39928, % error 0.09

Figure 54. Swaptions, ADP agent, second set of actions, avg. hr 48533, % error 0.10

Figure 55. Swaptions, ADP agent, third set of actions, avg. hr 47077, % error 0.10

Figure 56. Swaptions, Q-learning agent, first set of actions, avg. hr 30066, % error 0.20

Figure 57. Swaptions, Q-learning agent, second set of actions, avg. hr 49129, % error 0.11

Figure 58. Swaptions, Q-learning agent, third set of actions, avg. hr 44340, % error 0.11
Figure 59. x264, ADP agent, first set of actions, avg. hr 7.35, % error 0.29

Figure 60. x264, ADP agent, second set of actions, avg. hr 8.26, % error 0.27

Figure 61. x264, ADP agent, third set of actions, avg. hr 7.48, % error 0.28

Figure 62. x264, Q-learning agent, first set of actions, avg. hr 8.76, % error 0.42

Figure 63. x264, Q-learning agent, second set of actions, avg. hr 9.15, % error 0.47

Figure 64. x264, Q-learning agent, third set of actions, avg. hr 7.95, % error 0.39
<table>
<thead>
<tr>
<th></th>
<th>Best Window</th>
<th>ADP 1</th>
<th>ADP 2</th>
<th>ADP 3</th>
<th>QL 1</th>
<th>QL 2</th>
<th>QL 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>(8230000, 12345000) hb/s</td>
<td>5490257 hb/s, 0.23</td>
<td>10698401 hb/s, 0.16</td>
<td>9648170 hb/s, 0.11</td>
<td>9782523 hb/s, 0.14</td>
<td>10614396 hb/s, 0.12</td>
<td>7960482 hb/s, 0.12</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>(1.6, 2.4) hb/s</td>
<td>1.7 hb/s, 0.21</td>
<td>1.8 hb/s, 0.18</td>
<td>1.2 hb/s, 0.22</td>
<td>1.8 hb/s, 0.18</td>
<td>2.2 hb/s, 0.19</td>
<td>1.8 hb/s, 0.20</td>
</tr>
<tr>
<td>Canneal</td>
<td>(834000, 1251000) hb/s</td>
<td>979257 hb/s, 0.11</td>
<td>1125487 hb/s, 0.11</td>
<td>1075179 hb/s, 0.11</td>
<td>1121591 hb/s, 0.12</td>
<td>1130942 hb/s, 0.12</td>
<td>863815 hb/s, 0.10</td>
</tr>
<tr>
<td>Dedup</td>
<td>(3160, 4740) hb/s</td>
<td>5982 hb/s, 0.49</td>
<td>6386 hb/s, 0.57</td>
<td>2874 hb/s, 0.34</td>
<td>5956 hb/s, 0.64</td>
<td>7663 hb/s, 1.02</td>
<td>4938 hb/s, 0.47</td>
</tr>
<tr>
<td>Ferret</td>
<td>(11.6, 17.4) hb/s</td>
<td>12.0 hb/s, 0.18</td>
<td>15.0 hb/s, 0.25</td>
<td>13.4 hb/s, 0.15</td>
<td>13.3 hb/s, 0.17</td>
<td>15.5 hb/s, 0.28</td>
<td>14.6 hb/s, 0.23</td>
</tr>
<tr>
<td>Fluidanimate</td>
<td>(7.6, 11.4) hb/s</td>
<td>7.2 hb/s, 0.14</td>
<td>11.5 hb/s, 0.16</td>
<td>10.5 hb/s, 0.15</td>
<td>11.0 hb/s, 0.15</td>
<td>10.9 hb/s, 0.15</td>
<td>9.4 hb/s, 0.16</td>
</tr>
<tr>
<td>Raytrace</td>
<td>(5.6, 8.4) hb/s</td>
<td>6.8 hb/s, 0.14</td>
<td>7.2 hb/s, 0.17</td>
<td>6.7 hb/s, 0.17</td>
<td>6.1 hb/s, 0.19</td>
<td>7.0 hb/s, 0.14</td>
<td>5.8 hb/s, 0.19</td>
</tr>
<tr>
<td>Swaptions</td>
<td>(37960, 56940) hb/s</td>
<td>39928 hb/s, 0.09</td>
<td>48533 hb/s, 0.10</td>
<td>47077 hb/s, 0.10</td>
<td>30066 hb/s, 0.20</td>
<td>49129 hb/s, 0.11</td>
<td>44340 hb/s, 0.11</td>
</tr>
<tr>
<td>x264</td>
<td>(6.4, 9.6) hb/s</td>
<td>7.35 hb/s, 0.29</td>
<td>8.26 hb/s, 0.27</td>
<td>7.48 hb/s, 0.28</td>
<td>8.76 hb/s, 0.42</td>
<td>9.15 hb/s, 0.47</td>
<td>7.95 hb/s, 0.39</td>
</tr>
</tbody>
</table>
6.5 Comparison with Previous Work

In this section we propose a quantitative comparison between the results we obtained and the approach proposed in (4). In (4), the authors deal with the same problem of controlling the behaviour of an application instrumented with heartbeat calls through operating system level parameters. The benchmark suite used is, once again, the PARSEC suite.

There are two main difference in (4) with respect to this work:

1. The underlying decision making draws mainly from the control theory field (even though other approaches, such as heuristics, are explored).

2. The PARSEC suite is instrumented with Application Heartbeat (3).

Moreover, the results proposed in (4) were obtained on a completely different hardware with respect to the one we used.

In order to make an effective quantitative comparison we decided to repeat some of the tests in (4). We chose four different control mechanisms implemented there: a basic controller for the number of cores, a basic controller for the number of cores and the CPU frequency, a heuristic for the number of cores and a heuristic for the number of cores and the CPU frequency.

More details about these implementations can be found in the (4).

It is worth notice that the different instrumentation (application heartbeat and HRM) of the same PARSEC applications could make any confrontation of heart rates meaningless. We cope with this issue by only looking at application where heartbeat calls were placed in the same places in the source code, namely, Bodytrack, Fluidanimate and Raytrace (however, HRM
only allows integer numbers, while Application Heartbeats supports double precision floating point numbers).

Finally, we have to consider the fact that, while rational learning agents implemented in this work have reward functions assigning different values to different heart rate ranges, the controllers implemented in (4) have a “target values” computed, in this case, as the average value between the upper bound and the lower bound of the highest rewarded heart rate range.

Because of this, the error reported in the Table IV is computed in two different way for agents (in the following we only consider “trained” agents, i.e., learning agents that already went through the exploratory phase of learning and are now exploiting their knowledge) and the controllers from (4). For the agents, the computed error is the average distance from the highest rewarded heart rate range, for the other controllers the error is the distance from their target values.

In Figure 65 to Figure 66, we can see how heuristics behave, in general, poorly, while control theory approaches show performance by far superior in Bodytrack and slightly worse in Fluidanimate. However the control theory model does not seem able to cope with Raytrace as well as the model-free approach of Q-learning.
Figure 65. Bodytrack, comparison of a trained agent with control theory and heuristic approaches

Figure 66. Fluidanimate, comparison of a trained agent with control theory and heuristic approaches

Figure 67. Raytrace, comparison of a trained agent with control theory and heuristic approaches
### TABLE IV

**COMPARISON OF AGENTS, CONTROL THEORY AND HEURISTICS BY AVERAGE HEAR RATE (HB/S) AND ERROR (HB/S)**

<table>
<thead>
<tr>
<th></th>
<th>Best Agent (cores)</th>
<th>Basic Control Theory (cores)</th>
<th>Basic Control Theory (cores, freq)</th>
<th>Heuristics (cores)</th>
<th>Heuristics (cores, freq)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bodytrack</strong></td>
<td>1.744681, 0.345180 (adp 1)</td>
<td>1.455177, 0.547515</td>
<td>1.962514, 0.199694</td>
<td>1.968883, 0.212951</td>
<td>1.247088, 0.878063</td>
</tr>
<tr>
<td><strong>Fluidanimate</strong></td>
<td>10.909639, 1.309281 (ql 2)</td>
<td>1231.554970, 6131.753959</td>
<td>68.076128, 305.194433</td>
<td>6.408370, 3.850926</td>
<td>2.183806, 7.341655</td>
</tr>
<tr>
<td><strong>Raytrace</strong></td>
<td>6.500000, 0.872454 (ql 2)</td>
<td>1.187619, 5.812813</td>
<td>2.991823, 4.022371</td>
<td>3.025117, 3.988009</td>
<td>0.762630, 6.248515</td>
</tr>
</tbody>
</table>
In this chapter we come to some conclusions on the work done and its results. We also suggest what could be done next and our future research aims.

7.1 Final Considerations

At this point, we can summarize the work done. We started presenting the idea of autonomic computing systems, we explored properties and desired capabilities of such systems, then we went through some relevant artificial intelligence topics, relating to decision making and learning. After that, we presented several works in the scope of autonomic computing systems implementation (especially the sub topics of self-awareness and self-optimization).

Finally, we proposed our architecture for a self-aware/self-adapting/self-configuring autonomic element (at operating system scope) drawing from the previous works and artificial intelligence.

We then describe the implementation of this architecture and the tests we perform using the PARSEC benchmark suite (8).
Our experimental results show how the introduction of reinforcement learning algorithms in a system powered with sensing (in this case through the wrapper library of the kernel-space sensor HRM) and acting (in this case through libraries of functions that allows to tune processes affinity masks and CPUs operating frequencies) allows our system to learn action policies to adapt the environment to pre-specified goals, or, at least, to get as close as possible with the available actions.

We observe how different learning algorithms get to a correct policy in different amount of time, sometimes, the learning algorithm-action set combination does not allow the agent to control the PARSEC application in the desirable range However, it is a satisfactory result the fact that, for every PARSEC, we have at least one agent that eventually becomes able to control the application correctly.

7.2 Research Development

This work explored the possibility to implement artificial intelligence and reinforcement learning in order to bring autonomicity in an operating system. However, it also led the way several future developments we would like to study.

7.2.1 Further Testing

Of course, we would like, and we will, in the near future, perform more extended testing (that were not possible here only for time constraints) of the system built for this work (especially the learning capabilities integrated in AdaM). The experimental results provided in the previous chapter explore different learning algorithms (a model-based ADP algorithm and a model-free
Q-learning algorithm) and agents with different sets of available actions. The state description is always given by the only available sensor, HRM.

We plan on testing more learning algorithms (such as SARSA), more actions sets (including, for example, the ability to change the scheduling policy), several other exploration policies that could allow agents to learn faster, recover from wrong policies and adapt to later changes, we also plan on including new sensors to enrich the state description.

7.2.2 POMDPs

An other direction we want to explore is the use of more complex artificial intelligence frameworks, such as the ones intended for partially observable system. In this case, we would drop the use of learning and we would focus on a more complete representation/modeling of the system. The proper tool would be a POMDP.

Markov Decision Processes are defined on fully observable environments, the proper instrument to deal with conditions of noisy or partial observability, such as situations in which a sensors is not working properly or it is missing, are called Partially Observable Markov Decision Processes (POMDPs).

A partially observable Markov decision process is completely specified by a six element tuple:

A set of States $S$ containing the initial state $s_0$. One or more states can be “final”.

A set of Actions $A(s)$ for each state.

A stochastic transition model $T(s'|s,a)$
A **reward function** \( R(s, a) \) that returns a value for taking an action in a state.

A **set of Observations** \( \Omega \), all of the possible observations of the world from the point of view of the agent.

An **observation function** \( O(s, a, o) \) that returns the probability of making observation \( o \) after having taken action \( a \) and begin now in state \( s \).

The first four elements are the same ones defined for a standard Markov Decision Process (the reward function is different but this does not change the class of problems we can model by it).

In general, in the context of partial observability, taking in account the current percept and all the previous ones is not sufficient to make optimal decisions.

To have an agent that behaves correctly in a partially observable environment, we would want to consider also the history of all the actions performed by the agent until the current time and the history of all the previous observations.

In order to do so, it is useful to consider the concept of “belief state”. Belief states are “probability distributions over states of the world” (79).


16. IBM: Autonomic computing: Ibm’s perspective on the state of information technology.


45. Linux man pages.


### VITA

**Name**  
*Jacopo Panerati*

**Birth date**  
June 20, 1987

**Email**  
jacopo.panerati@gmail.com

**Mobile**  
+1 312 375 2447

### Education

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

_Enhancing Self Adaptive Computing Systems via Artificial Intelligence Techniques and Active Learning_

**2009–present**  

_A Reinforcement Learning Approach to Self-Adaptive Computing_

**2006–2009**  
Bachelor of Science in Computer Engineering, Politecnico di Milano, Italy.

_Software Model of a Turbine_

**2001–2006**  
High School Diploma, Liceo Scientifico C. Cavalleri, Parabiago (MI), Italy,  
*100/100*
Research

2010  *Comparison of Decision Making Strategies in Autonomic Computing*, TAAS Conference, Authors: Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, Alberto Leva

Experiences

2009  **Thesis Work**, C.E.S.I., Milan, Italy.

*Reference: Ing. Antonio De Marco*

Skills

*Coding*  C/C++, Java, Python, OpenGL, PHP, Bash scripting

*Engineering*  UML, JML

*Office*  Microsoft Office, LibreOffice, LaTeX, Beamer

*Databases*  SQL

*OSes*  Linux, Windows
Additional Information

*Languages*  English (TOEFL: 100/120, 2009), Italian

*Interests*  19th Century British and Russian Literature, Early 20th Century German Philosophy, Stand Up Comedy.