Table of Contents

[1 Introduction 3](#_Toc264278425)

[1.1 Motivation 3](#_Toc264278426)

[1.2 Contributions 4](#_Toc264278427)

[1.3 Publications 4](#_Toc264278428)

[1.4 Background 5](#_Toc264278429)

[1.4.1 Advanced Object- Oriented Concepts and Principles 5](#_Toc264278430)

[1.4.2 Autonomic Computing Paradigm and Autonomic Systems 6](#_Toc264278431)

[1.4.3 Intelligent Agents and Pervasive Computing 7](#_Toc264278432)

[1.4.4 Artificial Intelligence Learning, Knowledge Representation and Reasoning 8](#_Toc264278433)

[1.4.5 An introduction to Green Computing, Virtualization and Server Consolidation 9](#_Toc264278434)

[2 Problem Statement and Goals 11](#_Toc264278435)

[2.1 Problem Statement 11](#_Toc264278436)

[2.2 Problem Goals 11](#_Toc264278437)

[3 Related Work 13](#_Toc264278438)

[3.1 An overview of existing negotiation and bargaining solutions 13](#_Toc264278439)

[3.2 An overview of existing self-adapting systems 14](#_Toc264278440)

[4 Analysis and Design 17](#_Toc264278441)

[4.1 The context model 17](#_Toc264278442)

[4.1.1 Context Ontology Representation 17](#_Toc264278443)

[4.1.2 Policies 18](#_Toc264278444)

[4.1.3 Context Entropy 19](#_Toc264278445)

[4.1.4 State reward 19](#_Toc264278446)

[4.1.5 Inter-Independent Resources Group 19](#_Toc264278447)

[4.1.6 RAP instantiation for the self-healing system 22](#_Toc264278448)

[4.1.7 RAP instantiation for the self-adapting datacenter 22](#_Toc264278455)

[4.1.8 Load Distribution Strategy 22](#_Toc264278456)

[4.2 Reinforcement Learning Algorithm 23](#_Toc264278457)

[4.3 QoS-Energy Consumption Negotiation 25](#_Toc264278458)

[4.4 The Conceptual Architecture 27](#_Toc264278459)

[5 Implementation 31](#_Toc264278460)

[5.1 Architecture 31](#_Toc264278461)

[5.2 System Input 32](#_Toc264278462)

[5.3 System Output 33](#_Toc264278463)

[5.4 Architecture Implementation 34](#_Toc264278464)

[5.5 Ontology Representation Implementation 36](#_Toc264278465)

[5.5.1 Environment Ontology Implementation 36](#_Toc264278466)

[5.5.2 Datacenter Ontology Representation 36](#_Toc264278467)

[5.6 Agents Implementation 38](#_Toc264278468)

[5.6.1 Context Interpreting Agent (CIA) 38](#_Toc264278469)

[5.6.2 Reinforcement Learning Agent (RLA) 39](#_Toc264278470)

[5.6.3 X3D Agent Implementation 40](#_Toc264278471)

[5.6.4 GUI Agent Implementation 42](#_Toc264278472)

[5.7 Utility components 46](#_Toc264278473)

[5.7.1 Fuzzy Logic Negotiator 46](#_Toc264278474)

[5.7.2 Server Management Proxy 47](#_Toc264278475)

[5.7.3 Server Information Gathering Endpoint 48](#_Toc264278476)

[6 Testing and results 50](#_Toc264278477)

[6.1 Self-Healing Environment Test 50](#_Toc264278478)

[6.1.1 Overhead Test 52](#_Toc264278479)

[6.1.2 Interactive Test 52](#_Toc264278480)

[6.2 Self-Adapting Datacenter Test 54](#_Toc264278481)

[6.2.1 Simulated Datacenter Test 55](#_Toc264278482)

[6.2.2 Real Datacenter Test 58](#_Toc264278483)

[7 Conclusions 62](#_Toc264278484)

Chapter 1

# Introduction

This chapter describes the motivation behind the work that is presented in the rest of the book and the contributions this work brings to the selected field.

## Motivation

The ever-growing complexity of IT threatens its growth ]. The need to integrate heterogeneous environments into corporate computing systems and manage them approaches the maximum complexity a human can manage ]. As systems become more interconnected and diverse, system architects are less and less capable of anticipating component interactions, leaving those decisions to be made at runtime.

It is estimated that up to 50% of the total cost of ownership involves recovering or preparing against failures. Also, it has been estimated that IBM adds about 15.000 people per year in order to provide assistance to clients with complex platforms ].

The only option to the complexity problem is autonomic computing. By enhancing systems with autonomic properties the need for human intervention should decrease and the fault recovery time can increase. Also, the need for fault prevention decreases, as fault recovery can take place in real time without any costly human intervention.

Worldwide data centers electricity consumption accounts for almost 2% of the world production and is expected to overcome the 40% of Total Cost of Ownership of worldwide IT by 2012 ]. The US Environmental Protection Agency estimated that in 2006 the servers and datacenters power consumption accounted for 61 billion kWh (**!! De introdus in acronime**) , about 1.5 % of total U.S electricity consumption and for a cost of $4.5 billion ].

The same organization points out that the average data center is only 30% efficient, with 70% of the electricity lost due to inefficiencies of power and heat dissipation, along with powering cooling equipment ]. The environmental impact of datacenter expansion is of great importance, every server using 7,000 kWh of electricity and indirectly generating four tons of carbon dioxide emissions per year ].

This has lead to the need of finding environmental friendly methods for managing datacenters while maintaining performance. This new research trend has been called Green IT or Green Computing. The philosophy of Green IT is designing and using computer resources in an environmentally friendly way. This work has as aim reducing the overall number of servers used worldwide and further minimizing the number of powered on servers by applying server consolidation.

The main goal of this research is to develop a reinforcement-learning based multi-agent framework for building autonomic systems. This framework should be versatile enough to be used to build autonomic system, from a smart laboratory to an autonomic datacenter.

## Contributions

This project seeks to provide the necessary tool support for building and managing autonomic systems. Also, it should provide the necessary support for datacenter virtualization and consolidation. Another important contribution made by this work is the development of new negotiation methods between QoS and Power Consumption. There are both theoretical and practical contributions made by this project.

**Theoretical contributions**

* Develop a reinforcement-learning policy-based algorithm for autonomic self-healing environments.
* Adapt the previously mentioned algorithm for data center task management.
* Develop methods for negotiating between QoS (!!!!!!!!ACRONIME) requirements and Server Optimum Load Values (implying power consumption).

**Practical contributions**

* + - Create a multi-agent extensible framework for building and managing autonomic environments.
    - Apply the created framework for building and managing and autonomic environment.
    - Use the framework to create an energy-efficient datacenter hosted by a smart self-healing environment.
    - Develop tools for monitoring datacenter resources usage and environment sensors.
    - Develop 3D context representations for better system management.
    - Implement a QoS-Energy negotiation mechanism using Fuzzy Logic .
    - Find and evaluate appropriate technologies for implementing a self-adapting datacenter based on the framework mentioned above.

## Publications

The research conducted for this project has generated the following publications. The report chapters which include material from these publications are listed for each publication.

* **A Reinforcement Learning based Self-healing Algorithm for Managing Context Adaptation** ] presents a reinforcement learning based algorithm use for finding the optimum sequence of actions which enforce some user-specified policies. A proof of concept implementation is also presented, which uses an X3D (**!!! De pus in lista de achronimes)** representation of a smart environment in which a user click on an object to change its state. The algorithm detects if a user-specified policy is broken, searches for the best (having the highest reward) sequence of actions which “repairs” the context and executes them. (**!!!!** Material from this paper can be found in **chapter2onpage7,Chapter4onpage31andChapter5onpage47**
* **An Autonomic Algorithm for Energy Efficiency in Service Centers** ] (!!!!**De modificat daca nu I acceptat)** is an adaptation of the algorithm presented in the previous paper to handle datacenter task management. The old algorithm is still used for monitoring the datacenter environment. Both algorithms are used in a proof of concept implementation which uses a simulated datacenter in which tasks are dynamically added and removed, triggering consolidation actions.

## Background

This chapter provides an overview of the background theory in the area of Autonomic Systems, Artificial Intelligence, Self-Adapting Systems and Green Computing specific to the requirements of this project.

The following topics are discussed:

* Advanced Object-Oriented Concepts and Principles
* An introduction to the Autonomic Computing Paradigm and Autonomic Systems
* An introduction to Intelligent Agents and Pervasive Computing
* An overview of Artificial Intelligence Learning, Knowledge Representation and Reasoning
* An introduction to Green Computing, Virtualization and Server Consolidation

### Advanced Object- Oriented Concepts and Principles

There are high level concepts and principles used in this book that need to be clarified in order to understand how and why they were applied.

**General Responsibility Assignment Software Principles (GRAPS)** are well defined guidelines for assigning responsibility to classes and objects [ 33 ]. There are five main principles which must be enforced in any system which wants to benefit from a good design:

* **Information Expert** principle states that any responsibility should be assigned to the entity which has the necessary information to realize that responsibility. This principle helps to distribute responsibilities in a logical manner and makes further system modifications easier because each responsibility is realized by the entity which has all the information, not some other unrelated entity.
* The **Creator** principle is of great importance because the task of object creation is present in absolutely every object-oriented system. This principle states that an entity should be responsible of creating instances of another entity if it contains, uses closely or has the necessary information to create the instance.
* **Low Coupling** is a responsibility assignment principle needed by any system. By respecting this principle the impact of system change is greatly reduced, thus decreasing the cost of performing any change.

**Dependency Inversion Principle** is a principle which states that components should not depend on low-level components but on abstractions [ 35 ] . In a naive object oriented approach, the dependency would be on the classes located at the bottom of the class hierarchy tree (leaf subclasses). This type of dependency reduces the generality of the system which uses it and increases the coupling between classes. By changing the dependency from the low-level classes to the abstract super-classes or interfaces the dependency relation is *inversed*, the dependency now being to the top of the class hierarchy tree. This inversion allows for a greater amount of flexibility, each abstract dependency being able to use any concrete class that subclasses the abstract entity used in the relation.

**Separation of Concerns** is the process of separating a program in distinct features which overlap as little as possible [ 34 ].

**Open/Closed Principle** states that a system should be open for change but closed to modification. A more concrete definition is that the system should allow change without needing source code modifications to achieve that change [ 36 ]. In order to achieve this feature, other principles like *dependency of inversion*, *low coupling* and *high cohesion* must be respected.

### Autonomic Computing Paradigm and Autonomic Systems

**Autonomic Computing Paradigm** is a computing paradigm in which systems can manage themselves. This paradigm has been inspired by the human nervous system and has as goal creating applications that can manage themselves according to high-level goals given by humans ].

**Autonomic Systems** are self-governing autonomous systems. Such a system, given some goals to reach or policies to enforce will manage itself and take the appropriate actions to enforce the policies or reach the specified goals without any human intervention.

IBM frequently cites four properties of autonomic systems: Self-**C**onfiguration, Self-**O**ptimization, Self-**H**ealing and Self-**P**rotection ]. Such systems are referred to as Self-Star systems or **CHOP** systems from the four autonomic properties. IBM envisions that these properties will at first be treated separately but in time will merge into one general Self-Management concept.

**Self-Configuration** is desired as a property of every autonomic system because installing, configuring and integrating different vendor components into today’s systems is time consuming and error prone. Following high-level policies the system should adjust and configure automatically and transparently from the user’s point of view.

**Self-Optimization** need arises from the increasing complexity of today’s systems, involving hundreds of parameters which need to be manually tuned. A self-optimizing system will continuously seek methods of improving its performance and efficiency.

**Self-Healing** is very important because the error detection process performed by humans is inefficient and takes a long time. The ability of a system to detect errors and recover from them automatically would greatly improve the fault recovery time.

**Self-Protection** enables a system to detect attacks and recover from them. The automatic attack detection gives a faster response time, thus minimizing the damage done by the attack and the recovery time.

### Intelligent Agents and Pervasive Computing

Pervasive computing, also named as **everywhere computing** or **ubiquitous computing** ] is a computing paradigm in which information processing has been integrated in everyday life by means of small networked processing devices. These devices link communications and computing infrastructure to everyday life settings and commonplace tasks.

Agents are defined as anything perceives its environment through sensors and performs actions on that environment through actuators ].

#### Agent Environment

By their nature, the environments are roughly classified by ] in:

**Fully observable and partially observable environments:** As the name suggests, in fully observable environments the agent has all the needed information, while in partially observable not everything about the context is known.

**Deterministic and Stochastic environments**: In a deterministic environment the next state is determined entirely based on the current state and the action executed by the agent.

**Episodic and Sequential environments**: in an episodic environment there are independent episodes, while in sequential ones the next world state is always influenced by the previous one.

**Static and Dynamic environments**: In a static environment the world does not change while the agent thinks what to do.

**Discrete and Continuous environments**: Discrete environments have a finite number of states.

**Single-Agent and Multiple-Agent environments**: In multiple-agent environments agents compete for a set of resources.

#### Agent Types

Based on the complexity of the agent’s reasoning process the agents can be classified in 5 categories ]:

**Simple reflex agents:** Act only based on the current input data, ignoring the context history. The agent’s actions are determined by simple rules. Such an agent can succeed only in fully observable environments.

**Model-based agents:** This type of agents maintains an internal model of the world in order to be able to reason over partially observable environments. The model holds previously observed information that now is not observable anymore and reacts just as the simple reflex agent to input data.

**Goal-based agents:** Because in complex situations just reacting will not lead to the desired outcome this type of agent has appeared. These agents know some **goal states** and will try to reach those states.

**Utility–based agents:** Knowing about just the goal does not provide a measure of degree of how expensive is a set of actions the concept of utility is introduced. A **utility function** is used to map each an environment state to a desirability value, thus the agent being capable of comparing two courses of action and choose the one with the highest utility.

**Learning agents:** Are the most complex and can operate in unknown environments. These agents learn about the surrounding world as they go and continuously adapt themselves.

### Artificial Intelligence Learning, Knowledge Representation and Reasoning

#### Knowledge Representation

Any intelligent agent, even reflex agents use knowledge about the environment in their reasoning process. The process of knowledge engineering is a daunting one due to the complexity of the surrounding world.

**Ontologies** are the state of the art mechanism for knowledge representation. Ontology can be defined as a set of concepts from a particular domain together with the relationships between those concepts. These concepts are grouped in categories by the use of inheritance, thus providing a flexible and extensible means of representing the surrounding world.

#### Learning

The main idea behind learning is to use the gathered world data for improving the decision process, allowing an intelligent agent to improve its behavior trough study of its own experience. The goal of machine learning is for the intelligent system to be able to recognize complex patterns and make intelligent decisions based on them. For achieving this goal, machine learning has to borrow methods and concepts from other fields such as statistics, probability theory, data mining, pattern recognition and others.

Based on the type of feedback available, machine learning is classified in three categories: supervised, unsupervised and reinforcement learning.

**Supervised learning** is based on having the correct output for each input state. Mostly used in completely observable environments, in this type of learning the agent knows all possible outcomes from all possible actions and such it always knows what the best actions are.

**Unsupervised learning** is the opposite of supervised learning as it involves no knowledge of the system’s output for a given input. An unsupervised agent cannot learn what to do, because it has no information about what is desirable and what is not.

**Reinforcement learning** is a combination of the previous presented methods. The agent is not fully supervised nor is it left clueless. Instead, it receives a valuation function called reinforcement which indicates if a behavior is acceptable or not. The function value is the reinforcement, a negative value representing a penalty and a positive value an encouragement. Most important in this type of learning is the representation of the learned data and the accuracy with which the reinforcement function classifies the situation the agent encounters.

#### Reasoning

Reasoning in computer science can be regarded as the process of finding in the input data support for some concept. The most developed research area in this direction is in automated theorem proving algorithms, which are used in finding support for theorems based on the input data. Other research area as reasoning under uncertainty is of major importance in designing intelligent agent systems because the real world is not discrete and conclusions need to be drawn without knowing every outcome of every action.

### An introduction to Green Computing, Virtualization and Server Consolidation

Green Computing or Green IT is a computing paradigm that refers to environmental sustainable IT. All hardware and software components must use as little energy as possible and must have as small as possible impact on the environment. Another reason why this trend is important is because implementing its concepts reduces the total cost of ownership by reducing the energy cost.

#### Virtualization and Server Consolidation

A major problem in today’s datacenters is under usage of resources ] ]. Due to the lack of dynamic datacenter management based on the current or incoming load each datacenter must have all servers online in the event that the traffic increases .Also, being prepared for the largest traffic possible leads to the need of having a large number of servers. These two facts combined generate a problem called **server sprawl**, a situation in which multiple, under-utilized servers take up more space and consume more resources than can be justified by their workload.

**Server Consolidation** () has as goal solving this problem by finding means of reducing the number of idle servers. There are several methods for server consolidation, some based on more powerful servers, some on virtualization.

**Virtualization** is a technique for running several operating systems on a single system. By employing virtualization many servers can be transferred to dedicated virtual machines and run on a more powerful system (Fig 1), thus increasing server consolidation. In virtualized environments, the virtual machines run on a “virtual machine OS” called a **hypervisor** which manages the resource allocation and distribution among the virtual machines. A major advantage of this approach is that a virtual machine’s resources can be modified depending on the load. Also, with modern hypervisors, virtual machines can be migrated from one server to another without any visible downtime to increase server utilization or in the event of a hardware failure.

Figure 1.1: Server Consolidation trough Virtualization

#### Datacenter Load Distribution

**Load skewing** is a datacenter load distribution technique which involves several tasks. The entire datacenter load is fitted onto as few servers as possible. One or more servers are kept as **tails** in order to accommodate future workload. The rest of the servers are sent to hibernation.

**Throttling** is the mechanism of reducing the performance of underused hardware components and thus reducing the energy consumption. This mechanism is very useful in datacenters where the workload is evenly distributed among servers.

Chapter 2

# Problem Statement and Goals

## Problem Statement

The purpose of this book is to present a framework for building autonomic systems based on mobile intelligent agents which uses ontologies for context representation and policies for goal description. This solution must provide a general extensible platform for building context-aware self-adapting systems applicable to any field. The framework should provide a uniform mechanism of data gathering, context representation and policy enforcement. The policy enforcement engine should detect when the context is in a state which violates at least one policy. It should use a reinforcement learning algorithm to search for the best course of action which brings the context back in an accepted state. The expected reward of being in the resulting state must be the highest between reachable acceptable states. The reward mechanism must be flexible and should capture accurately the differences in severity between different unacceptable states based not only on the number of broken policies, but also on the distance to the closest acceptable state and the number of policy sub rules which are violated. For completing the environment management, the proposed framework must provide a flexible and extensible mechanism for actuator management.

## Problem Goals

1. Create a suitable context representation
2. Build a context management suite
3. Apply the context management solution to a self-healing environment
4. Extends and adapt the context management solution for datacenter management

In order to achieve this goals there are some important sub-goals to be met.

1. **Create a suitable context representation**
   1. Study the RAP ]context model
   2. Discover the context object types and their relationships
   3. Build a context ontology representation using Protégé ].
2. **Build a context management suite**
   1. Create an information gathering mechanism:

* Create an information gathering module for the System Under Test(SUT) endpoint
* Create a context information consumption mechanism for querying the SUT information gathering modules.
  1. Create an extensible actuator management mechanism:
* Insert the actuators in the context ontology representation
* Design a self-adapting algorithm which can handle in real time actuator list modifications.
  1. Build a logging mechanism for better context monitoring and debugging
* Create a PDF log writer and a log display window
  1. Build a distributed context management framework based on mobile agents
* Study intelligent mobile agents
* Define agents and their behaviors
* Choose an inter-agent communication mechanism
  1. Build a 3D context representation:
* Study the available 3D technologies
* Find a synchronization method between the 3D representation and the real context
* Make the representation interactive: interactions with the 3D context must generate actuator actions.
  1. Provide additional visual system information:
* Provide a real-time plot representing the algorithm running time
* Provide a visual representation of the context policies, sensors and their values

1. **Apply the context management solution to a smart environment**
   1. Create web services for the smart environment sensors
   2. Define policies, actuators and associated actions
   3. Test the context management solution under context patterns
   4. Extensively test the system using the interactive 3D representation
2. **Extends and adapt the context management solution for server cluster management**
   1. Create web services for interacting with the server operating system
   2. Adapt the context representation

* Split the policies into two categories : Energy policies and QoS policies
* Augment the context representation with information particular to datacenters
  1. Find techniques for improving the energy efficiency in datacenters
* Find a mechanism for dynamic task migration
* Find a mechanism for improving server consolidation
  1. Study datacenter management technologies
  2. Provide system information visually:
* Adapt the existing 3D representation
* Implement server resource monitors
* Create a visual representation of the tasks(both deployed and undeployed) for better task tracking
  1. Test the system on a real world server cluster
  2. Develop a workload generating tool for extensive system test.

Chapter 3

# Related Work

The related work this paper is based on can be classified into two categories:

* Existing negotiation and bargaining solutions
* Existing self-adapting systems

## An overview of existing negotiation and bargaining solutions

An improvement to server consolidation is the use of negotiation techniques to find a tradeoff between power consumption and QoS. Although QoS requirements are of most importance, in some case a tradeoff is needed due to hardware failure or even unjustified power consumption. For example if a decrease in 5% of CPU requirements for the entire datacenter would allow, after virtual machine rearrangement, for one server to be turned off or send to low power state, then a negotiation technique can be used to find the best value to decrease the CPU for each virtual machine.

The research area of distributed negotiation or multi-agent negotiation has a lot of work associated to it. From this work is clear that negotiation is an important issue in almost any distributed or intelligent system, being present from web services negotiation to grid resource allocation and with the help of this paper, in datacenter consolidation efforts.

Web service discovery and invocation benefit from negotiation as described in ], where a tradeoff between QoS and Cost of Service is achieved trough exchange of less desirable tokens with more desirable ones between parties. As presented in ], very important in web service discovery is the process of finding the most appropriate Web Service providers for a specific Web Service requestor. For being able to conduct real world business by automated web service composition, a negotiation mechanism must exist and it must ensure an optimal “deal” for both sides. This approach introduces *logrolling*, a negotiation technique used mostly in politics, where one person can trade his vote in the exchange of a vote received for his law proposition. The negotiation framework is based on tokens, split in two categories: Quality of Service (QoS) tokens and Cost of Service (CoS) tokens. Logrolling is combined with a token-based negotiation framework augmented with token weights representing each token’s importance. Using this solution, the provider and supplier exchange tokens until both sides reach a consensus, a state in which the utility functions of both parties are the same.

Another approach to multiple-party negotiation is presented in ] under the form of a logic programming framework. This approach, instead of negotiating the existing proposal focuses on creating a counter proposal, ignoring the process of deciding on the utility of a proposal. The system is centered on a knowledge-base which contains all the necessary information about creating a counter-proposal. The proposal is generated using Abductive Logic Programming ], capable of dealing with the unknown goals of the other negotiating party. In case the counter-proposal is rejected, another proposal is generated from the previous one by relaxing the knowledge-base search criteria. This automated negotiation approach solution can be built on top of existing answer set solvers, thus providing a clear path to concrete results.

The research work in the field of multi-issue negotiation can be split into three categories after their understanding of the best negotiation result: minimum loss, maximum gain, and the more general utility function maximization. An approach that fits in the first category is ] which describes negotiation as searching for envy-free states in multi-agent environments. The negotiation process is viewed as a resource allocation one for better understanding of the presented concepts. Given that each agent involved in the resource allocation process has a valuation function to indicate its resource preference, this work focuses on proving that in an envy-free state the resources are efficiently distributed. It proves that envy-free states can be reached if they exist and that resource allocation efficiency and envy-freenes are compatible.

] fits in the second category, searching for joint gains as negotiation result in multiple independent issue negotiation. This approach focuses on “creating value” instead of minimizing loss. For coping with the tendency of agents to hide their intentions in order to obtain as much profit as possible from a trade, an impartial mediator is introduced. Agents disclose secret information to the mediator, which based on that information, tries to achieve a fair deal. Another key element of this solution is that it computes a Pareto-optimal set of outcomes (outcome in which no improvement can be made for a party without worsening of the outcome for the other party). The Pareto-optimal set is further inspected for finding fair situations in which the inter-agent resource distribution is appropriate. Other than providing a generic multi-issue multi-agent negotiation framework, ] also presents utility maximization methods and compares them with respect to their manipulation susceptibility for use in environments where agents tend to misrepresent their utility.

An improvement over existing negotiation techniques is brought by ] which describes a involving multiple interdependent issues, as encountered in many real-world scenarios. A Distributed Mediator Protocol is defined for finding Pareto-optimal agreement points. In order to have an efficient solution, a genetic algorithm is used to find multiple Pareto-optimal agreement points in the nonlinear negotiation space. The genetic algorithm is compared with two other search methods: simulated annealing and hill climbing. Another algorithm called Direct Search is defined which maximizes Nash products ] without finding Pareto-optimal agreements. The Direct Search algorithm is compared against the Distributed Mediator Protocol and an approximated fairness concept is introduced for maximizing Nash products.

(!! DACA SE POATE SA SCOT PAPERU ALA CU FUZZY SA BAG PE AICI K AM NEVOIE DE EL SA DEMONSTREZ K I NOU CE AM FACUT)

## An overview of existing self-adapting systems

The work in the area of self-adapting systems is usually centered on the use of ontologies for representing context information and policies for representing goals. This leads to the need of having a reliable mechanism for gathering data from the surrounding environment and representing it in a manner that supports reasoning. Such a mechanism is presented in ] under the form of an event-driven publish/subscribe architecture for data gathering, processing and event creation. Events are created based on the input data and are used by subscribers to monitor different areas or activities within the surrounding context. The events are fed to the event processing system through a series of event streams. Adding semantical information to an event, each event is assign to a context. This approach also defines a mechanism for fusing similar events from the same context and combining the information given by the last event with the old information held in the knowledge base. Aside from event processing, an alert mechanism is presented, used to inform about specific situations detected based on generated events. The mechanism is applied to real world monitoring situations, like video monitoring of a room. The major contribution brought by this work is the event contextualization mechanism used to map events to contexts, thus providing a semantic event space which can be further extended and applied to various domains.

Another approach to context information gathering is presented in ]. This work focuses on the self-configuring aspect of context-aware systems in the area of pervasive computing ]. It provides a dynamically reconfigurable fault tolerant context management system. The system is based on a context model containing both information about the required context information and context information metadata. The use of sensor description standards is advocated, as they support opportunistic discovery and integration of sensors and thus adding flexibility to the overall system. By relying on the IEEE 1451 smart sensor interface and SensorML ] sensor description framework , this approach allows smart sensors to advertise and describe themselves to higher level management systems. The proposed approach is testes for a rescue crew scenario having a central management unit and several sensors mounted on each person. Testing results demonstrated the adaptability of the system from using sensor description standards.

One important research direction in context-aware system is in Multi-Agent Systems. Such a system is presented in ] for management of power and performance in datacenters. The purpose of this paper is to demonstrate practically that agents can be used for implementing a coherent automated datacenter management system. The system is centered on three agents: Performance Agent, Power Agent and Coordination Agent. The Performance Agent is responsible for load distribution among servers. The Power Agent is responsible for power consumption monitoring and setting power caps. Also, this agent uses a drastic power consumption reduction mechanism by turning off and on servers depending on the datacenter workload. Finally, the Coordination Agent handles the communication between the other two agents and uses predefined utility functions and policies in sending control signals to the system. The presented system was tested on a datacenter hosting IBM blade servers running Linux. Although the experimental results are promising, the technology is not there to help in implementing such a radical power management. A server running Linux takes 5-10 minutes to start after being turned off, an amount of time which makes it difficult to implement such an approach in real-world datacenters.

Under the pressure to make computing eco friendly an important research branch in self-adapting systems is creating energy-aware systems that are capable of reducing power consumption and maximize resource usage. One such approach is ] , which presents a dynamic load management system for virtualized datacenters. Employing virtualization, this solution allows a single server to be shared by multiple services, thus improving server consolidation and resource utilization. Also, virtualization allows for on demand task resource allocation based on the datacenter workload. For maximizing the energy consumption, this solution, as the one presented before exploits the idea of turning the servers on and off depending on the datacenter load. Although turning off servers has the visible advantage of lower power consumption, the process of waking up a server implies a large amount of time, as discovered and described in ]. In order to minimize the impact of the server wake-up time, a limited look ahead control mechanism is introduced. Also, the server switching costs are considered. From the experimental results, using server virtualization together with a lookahead needed for anticipating datacenter state results in an average of 26% power reduction while maintaining QoS requirements.

Very important in self-adapting systems is the system’s ability to learn the proper actions that solve some problematic situation. One school of thought is focused on applying reinforcement learning algorithms to smart environments. Reinforcement Learning advocates view this type of learning as a more “human” way of searching for solutions. Also, it seems to be more natural to think of goals as states with high rewards. Such an approach is presented in ] which demonstrates that reinforcement learning can be used successfully in autonomic systems. The reward is computed based on the multi-attribute system state and represents how desirable is for the system to be in that particular state. The selected learning algorithm is State-Action-Reward-State-Action (SARSA) and it is compared with a Goal-Action Attribute Model technique ] and. The solution is tested using a series of simulated models and the result is that reinforcement learning is feasible for the management of autonomic systems, but future research is needed because this type of approach does not provide a high performance solution

Chapter 4

# Analysis and Design

The system’s core algorithm and associated concepts where developed by me and Copil Georgiana, author of ], for the **CONSENS** research project ] . The results of the research in the context of this project are described in ].Due to these facts, the **reinforcement based self-healing algorithm** and associated concepts descriptions will appear both in this book and in ]. The entire andsubchapters are joint development effort, elements contained in it being also described in ].

In this chapter both the core components of the autonomic system framework and the design decisions needed for applying the framework to build a self-healing laboratory and a self-adapting datacenter are presented.

## The context model

The context model used by the self-adapting solution presented in this book is based on the RAP (Resources, Actors, Policies) model presented in ]. This model was chosen because is general and powerful enough to be used in any self-adapting system. Any context can be represented as a collection of Resources, Actors that interact with those resources and a set of Policies which control the state of the resources and actors and of their interaction.

### Context Ontology Representation

The core ontology representation only extends the Resources element of the RAP model with Sensor and Actuator concepts. represents the core ontology structure. The distinction between resources is needed as the system needs to know what element returns context information and what element can influence the context. This is important because the system iterates trough actuators attached to a sensor which does not respect some policy and tries the available actions in order to see if some improvement over the current situation can be made.



Figure 4.1: RAP Ontology Representation

Because the datacenter context needs more specific information, another ontology representation is defined based on the RAP model and used together with the previous basic representation. The datacenter ontology representation is illustrated in . The previous representation is used for environmental representation and the next ontology is used to represent the datacenter physical and software components. Also, due to the fixed number of actions possible in a datacenter, the generality of the representation is reduced by creating a concrete class for each possible action. The reduction in generality is motivated by an increase in efficiency by eliminating the search for all possible actions at each step in the action selection algorithm.



Figure 4.2: Datacenter Ontology Representation

### Policies

The policies enforce some high-level conditions and consist of a set of acceptable sensors values. In order to have a fine-grained view over any context situation and be able to differentiate between the severity of two different context situations, weights are used. Each policy has a weight associated to it which represents the importance of the policy in the overall context. Furthermore, each resource associated to a policy has a weight representing the importance of the sensor in that particular policy. The formal policy definition is defined in .

Equation 4.1: Policy Specification

|  |
| --- |
| [, |
| , where:  - : function which computes value of Resource i  - **:** acceptable range of values or Resource j |

### Context Entropy

For understanding the concept of context state entropy the state must be defined. A context state is a snapshot of the execution environment taken at a specific moment in time.

For evaluating the degree of context state deviation from the acceptable value the concept of context state entropy is used ]. The context state entropy is used to represent the desirability of the system to be in a particular state. Based on the definition of the policy, the entropy is defined as the product between the policy weight and the sum of products between the associated resources weight and their distance from the acceptable value. The Entropy formula is defied in .

Equation 4.2: Entropy Formula

|  |
| --- |
|  |
| , where:  - : entropy  - **:** the weight of i-th policy  - **:** the importance of resource j in policy i  - **:** is a value marking the degree to which resource j respects policy i. |

### State reward

For the self-healing environment management algorithm the reward is represented by the inverse of the entropy. The states with the highest reward are the states with the smallest entropy, as resulting from .

Equation 4.3: Reward function

|  |
| --- |
|  |
| , where:  - R**:** the state reward  - **:** the state entropy |

For the self-adapting datacenter management algorithm the reward is computed using a more fine-grained formula described in ].

### Inter-Independent Resources Group

The time required to find the best sequence of actions that brings the system in an accepted state depends on the number of policies and on the number of resources attached to a policy. For improving the performance of the system the concept of Inter-Independent Resources Group (IIRG) is introduced. In order to understand the IIRG concept the dependency relation must be defined first. A resource is dependent on another resource when a change in the value of one of the resources triggers a change in the value of the other resource. Furthermore, the dependency relation is of two types: direct dependency and indirect dependency. Direct dependency between two resources and occurs when a change in the value of triggers a change in ’s value. Indirect dependency between two resources and occurs when a change in ’s value triggers a chain of changes in the values of several resources () which eventually affects . Both dependency types are formalized in .

Equation 4.4: Dependency Relation

|  |
| --- |
|  |
| , where:  - : Resource X  - : implies, triggers  - **:** dependency between Resource X and Resource Y |

The dependency relation is represented under the form of a dependency matrix with has as entries the length of the dependency between the resources (1 for direct dependency and path length +1 for indirect dependency). A “0” in the dependency matrix means no dependency. The dependency matrix representation was chosen because it allows for fast direct dependency verification and also it provides support for indirect dependency checking by allowing the use of graph cycle detection algorithms for dependency chain detection. A dependency matrix example is shown in .

Table 4.1: Resource Dependency Matrix example

|  |  |  |  |
| --- | --- | --- | --- |
|  | R1 | R2 | R3 |
| R1 | 0 | 0 | 0 |
| R2 | X | 0 | 0 |
| R3 | Y | 0 | 0 |

From the dependency relation defined above it results that two resources are inter-independent if there is no direct dependency between them or if the length of the indirect dependency chain is over a certain limit. The certain limit statement is included because the influence of the resources can drop as the dependency chain increases in length, reaching a state where the dependency is negligible and thus can be ignored. By separating the resources in IIRGs solutions for context repair can be searched in parallel over the IIRGs, thus reducing the system’s search space and improving the running time. The pseudo code of the IIRG extraction algorithm is presented in .

|  |
| --- |
| Listing 4.1: IIRG Extraction Algorithm |
|  |

For a better understanding of the IIRG extraction process, Figure 4.3 presents a trace on an example where it is assumed that the resources weights follow this rule : weight(R2) >weight (R3) >weight (R1).



Figure 4.3: IIRG Extraction Algorithm Trace

### RAP instantiation for the self-healing system



For using the RAP model in implementing a self-healing environment, the domain model elements must be mapped onto real world entities. The mapping is represented in .

Each sensor can have associated one or more actuators which have an influence on it. Each actuator has at least one action defined, each action having its effect represented by a **+ value (increment action), - value (decrement action)** or **value (set value action).**

Table 4.2: Self-healing environment RAP mapping

|  |  |  |
| --- | --- | --- |
| <R,A,P> entity | Self-Adapting context entity | Details |
| Resources | **Sensors** | * Smart room sensors |
| **Actuators** | * Components capable of influencing the environment |
| Actors | **Actions** | * Each actuator has associated actions |
| Policies | **User specified policies** | * Acceptable sensor values combinations for the environment |

### RAP instantiation for the self-adapting datacenter

While the previous RAP instantiation is still useful in the self-adapting datacenter for environmental control, a new mapping is needed for datacenter workload management. Due to the reduced number of possible datacenter actions (excepting environmental control) the new mapping is more concrete and includes five concrete action instances: Deploy task, Move task, Send server to hibernate, Wake up server. The reduction in representation generality is compensated by the increase in representation efficiency, reducing the action search to a set of five predefined instances. The RAP mapping for the datacenter scenario is presented in .

Table 4.3: Self-Adapting Datacenter RAP Mapping

|  |  |  |
| --- | --- | --- |
| <R,A,P> entity | Self-Adapting context entity | Details |
| Resources | **Servers** | * All datacenter servers |
| Actors | **Actions** | * Deploy task * Move task * Send server to hibernate * Wake up server |
| Policies | **Energy policies** | * Servers green performance indicators ranges |
| **QoS policies** | * Task requested resources ranges |

### Load Distribution Strategy

For optimizing the energy consumption for a datacenter is not sufficient to have an efficient workload distribution strategy. The main reduction in datacenter energy consumption is obtained by sending to a low power state (hibernation or sleep mode) as many servers as possible. This leads to the problem of finding a mechanism for reducing the system response delay introduced by the server wake-up time. In a real-world datacenter scenario, when a task is required to be run the task can’t wait for a server to power up. In order to solve this problem server skewing with tails is employed. This approach keeps one or more servers are kept online for accommodating future loads based on workload predictions. Also based on this prediction a **load threshold** is computed for each tail and when that threshold is reached a new tail server is brought online. The idea behind the threshold is to be computed dynamically based on workload prediction. For maximum efficiency, the time it takes for a server to wake up should be the time it takes for the tail server to become fully used after the load threshold is reached. An example of a situation in which the load threshold is reached, thus triggering a wake up of another server as tail is presented in .



Figure 4.4: Server Skewing with Tail

## Reinforcement Learning Algorithm

For finding the best sequence of actions to bring the system in an acceptable state an algorithm based on reinforcement learning was developed. The advantage of basing a reinforcement learning based approach is the generality provided by this type of learning. Any real world situation can be represented in terms of <action, reward> pairs. Also, by manipulating the expected reward the mobile agent that uses this learning type can be guided to take any desired course of action without actually specifying the actions he needs to take. The adaptability of such an intelligent agent is a major advantage in building a generic platform for building autonomic systems.

The desired context states are described using policies. Based on these policies the algorithm computes the entropy value for the current context state using the formula defined in 4.1.4 State reward. If the entropy value is above 0 (or above a user-defined threshold) the search for the repairing sequence of actions is triggered. For each broken policy, the algorithm takes each broken resource and tries to change its value in an acceptable one by executing actions on associated actuators. Each associated action executed is simulated, the resulting state’s entropy is computed and the state is placed in the states priority queue, sorted by entropy. The next state to be expanded is the top of the queue, which holds the best state found so far. The pseudo code of the generic reinforcement learning algorithm is presented in .

|  |
| --- |
| Listing 4.2: Reinforcement Learning Algorithm |
|  |

Due to the generality of the reinforcement learning approach, the previously described algorithm can be applied easily both to a self-healing smart environment and to a self-adapting datacenter. Also, the algorithm expands the search space in a breadth-first based manner, but always expanding the node which has the highest reward associated. This search mechanism allows for finding the best solution in a less amount of time. In the reinforcement learning process is represented.



Figure 4.5: Self-Healing system Reinforcement Learning Process

Due to the real-time performance required from real world datacenters, for applying the algorithm to a datacenter, the algorithm action search must be stopped at any given moment in time and the best action sequence found so far to be used. This constraint comes from the real-time performance demands of a datacenter. In the case that a very complex scenario with an increased number of servers and a large number of tasks that need to be run takes place, the algorithm might take too long to find the best solution. In this case the best solution found so far needs to be used in order to respect the QoS requested from the datacenter. The new flow of the algorithm is presented in .



Figure 4.6: Datacenter Reinforcement Learning Process

## QoS-Energy Consumption Negotiation

In the real world sometimes situations in which some policies conflict can appear. For example one policy should request a value or range of values for some resource and another policy could request another range of values for the same resource at the same time. In this case the conflict needs to be solved somehow. A good conflict solving mechanism is negotiation, in which each party involved in the dispute gives up something until everyone is satisfied and a consensus is reached.

The negotiation problem was studied intense for the self-adapting datacenter implementation case because here the conflicting policies problem is the most pertinent. As described in , each server has associated green performance indicators. These indicators specify the optimum server load for maximizing the performance-per-watt ratio. The performance indicators are specified as ranges. For example, a server could have an optimum CPU load of 60%-80%. The task resource requirements are also specified in ranges, for example for CPU 500 MHz-800 Mhz. A situation in which the task requires more from the server that he is willing to give can appear easily. In such a situation choosing to enforce one policy or another based on some policy importance is not the best choice because an energy efficient datacenter which respects QoS requirements is needed and this implies that both energy and QoS policies need to be enforced.

In a situation where negotiation is required is presented. In such a situation the maximum resources required by a task can’t be allocated, but there is room to fit the task in the required resources range. A basic approach would be to allocate the minimum resources requested by the task. But this might lead to situations in which the maximum optimum server load is not reached, but some task performs in a degraded mode with the minimum resources needed to run. For avoiding this kind of situation, a tradeoff mechanism is needed. The tradeoff solution presented here follows the reinforcement approach taken to system management in this book.



Figure 4.7: Situation which requires negotiation

**Desirability** is defined here as the satisfaction of the system to be in a particular state. This concept can be applied both to server optimum load indicators and to task requested resources range. Desirability information can be added in the context by using a bidimensional function, one dimension for desirability and the other for the actual value. Such a function is presented in . The desirability is represented in the Y axis, while the corresponding values are on the X axis.



Figure 4.8: Desirability-Based Negotiation

The negotiation is done by taking the requested task range and the available server range and checking if the two ranges overlap. If so, the overlapping range is extracted. A desirability function is assigned to the overlapping range based on the QoS and Energy Consumption importance. The result is another bidimensional function. In the desirability of the task is assigned to the overlapping range in order to achieve a greater QoS. Finally, the negotiation result is the **Center Of Gravity** of the area given by the previously generated bidimensional function using .

Equation 4.5: Center Of Gravity Formula

|  |
| --- |
|  |

## The Conceptual Architecture

The system’s architecture is a multi-agent one. The reason for choosing such architecture is its mobility, extensibility and separation of concerns it provides. These reasons make multi-agent systems very good candidates for implemented distributed systems. After establishing the agent interaction mechanism, each agent can be developed separately and also it can reside anywhere in the distributed system. Also, any agent can change location inside the distributed system without affecting the system’s functionality.

Having an agent-based architecture, the system consists of six agents: Context Model Administering Agent (CMAA), Context Interpreting Agent (CIA), GUI Agent, X3D Agent, Task Management Agent (TMA) and Reinforcement Learning Agent (RLA), which is the most important agent and implements the core functionality needed for this autonomic system building framework. The CMAA reads an xml context description file and loads the information into the shared ontology. The context description file contains xml definitions of ontology instances. CIA periodically queries the context sensor data and refreshes the ontology context information. The GUI Agent is responsible for the graphical representation of the system status: system log, sensor values, etc. RLA is the most important agent. It detects when the context is broken and searches for actions to bring it back in an acceptable state using a reinforcement learning based algorithm applied on the common context ontology representation. The X3D Agent is responsible for the 3D interactive context representation. The 3D representation is sued both for environment simulation and for environment control, allowing the user to interact with the virtual environment and test the autonomic mechanism which manages it. The TMA is used only in the self-adapting datacenter extended version of the proposed framework and is responsible for dynamic task creation. This agent is very important as it allows the user to create a task at runtime and such it provides a good testing mechanism and also brings more dynamicity to the system. The agent-based conceptual architecture is illustrated in Figure 4.9.



Figure 4.9: System Conceptual Architecture

One important issue in any multi-agent systems is responsibility assignment. The responsibility assignment process was guided by the General Responsibility Assignment Software Principles (GRASP) [ 33 ]. Conform to the *Creator* principle, the CMA Agent creates all the other agents because it holds the necessary information needed to instantiate the agents. The CMA agent loads the ontology description, creates ontology representation models and during the agent instantiation process it sends those models to the agents. The GUI Agent respects the *High Cohesion* principle as it is concerned only with displaying information received from the other agents. If the *Information Expert* pattern would have been used here it would have violated the *High Cohesion* principle, because each agent would have had also the responsibility of displaying the context information which it contains. The same principle, *High Cohesion*, was applied when assigning the responsibilities to the other agents. The other GRASP principle left unmentioned, *Low Coupling,* is respected implicitly by using a multi-agent architecture. Because the agents communicate only trough a well defined mechanism over the network, without direct method invocation, any agent can be replaced without affecting the other agents as long as it returns the expected data.

Another important issue in multi-agent systems is the agent interaction mechanism. After assigning responsibilities between agents it must be decided which agent communicates with which agent and what type of information to the agents exchange. In Figure 4.10 agents interactions are represented. The Context Interpreting Agent captures data from the environment sensors, populates the context ontology representation with that data and informs the X3D Agent about the modification and sends the new data. This notification is needed as the X3D agent does not use the ontology representation. Instead, it receives pairs of <sensor name, value> from the CIA and RLA and uses that information to refresh the 3D representation. This mechanism allows the X3D Agent to be used independently on the underlying ontology representation, bringing a needed plus in system flexibility. By adhering to the *Low Coupling* and *High Cohesion* principles, the X3D Agent minimizes the effect a change in the underlying ontology has on the system.



Figure 4.10: Agents Interactions

The Reinforcement Learning Agent (RLA) in turn sends notifications to the GUI Agent and X3D Agent in order to refresh the displayed data for both agents. The GUI Agent needs messages sent to it because it handles the logging of the RLA actions and it cannot have that information retrieved by any other means. The X3D Agent receives information from the RLA which is needed for synchronization of the 3D representation with the underlying ontology used by the RLA.

In turn, the X3D Agent notifies the RLA when some interaction with the 3D objects takes place. Because the X3D Agent does not interact with the ontology used by the RLA, it cannot change the ontology values directly so it must inform the RLA of what interaction has place. This is conforming to the *Information Expert GRASP* principle because the entity which has the information needed to execute an action executes the action. In this case this expert entity is the RLA and after receiving the notification from the X3D agent it enforces the action on the environment actuators

The Task Management Agent (TMA) interacts with the RLA because it does not have the possibility of creating a physical datacenter task. So it sends a create task notification together with the additional information needed about the new task to the RLA. The RLA creates an ontology instance for that task and executes a create task action on the datacenter.

Chapter 5

# Implementation

The current chapter focuses on the development and implementation of the proposed agent- based context management framework. It gives a closer look at the technologies used to implement the context management components presented in the previous chapter, and then it explains how these technologies were used to obtain the proposed goals.

## Architecture



Figure 5.1: Architecture

Only the components developed by me (highlighted in with green) will be presented in detail. The other components are detailed in ] by their creator and will be mentioned and properly referenced when needed in order to maintain a certain system description mental flow.

The “Environment Self-Healing Module” (highlighted half with blue and half with green) was developed by me and the Copil Georgiana, the author of ] for the CONSENS research project ] and as such it is a joint development effort, also described in ].

## System Input

For any knowledge-based system to be able to reason about the surrounding world it is necessary to have a mechanism of describing that world. The selected mechanism is **Ontology Representation**.

The **context** is described using the two previous ontology representations ( and ). Both the datacenter and environment ontology representations are built using Protégé ] and populated with instances. The instances represent real world physical elements. A closer look of the ontology representations used by this architecture is presented in Figure 5.2.



Figure 5.2: System Ontology Representation

The **system acceptable states** are defined by policies specified under the form of **SWRL (Semantic Web Rule Language)**  ] rules. By specifying the policies in SWRL the system uses **Pellet** ] OWL2 Reasoner for policy evaluation. The SWRL rules evaluation triggers at each change in the ontology variables, providing a reliable mechanism for context evaluation. An example of a SWRL QoS Policy definition is presented in Listing 5.1. The SWRL rule is simplified in the listing mentioned previously in order not to overload the representation. In the example for each property a value is specified, when in the real SWRL representation there is a value range. An example of a SWRL rule definition is illustrated in Listing 5.1.

|  |
| --- |
| Listing 5.1: SWRL QoS Policy Example |
| J:\cercetare\paper suceava\deliverables\source images\swrl_QoS_policy.png |

The SWRL rule example from Listing 5.1 enforces several conditions for *Task* with *name* “Task\_1”:

* Task HAS-PROPERTY *associatedInfo*
* Property *associatedInfo* has property *Received*
* Property *Received* has properties *cores, cpu, memory and storage*
* Property *cores*  has value 1
* Property *cpu*  has value 3000
* Property *memory*  has value 2048
* Property *storage*  has value 200

If the conditions presented above are met, the *QosPolicy\_1* instance has its *respected* property set to true. Otherwise the *respected* property is set to *null*.

The **world information** is accessed using **ASP.NET Web Services** ]which provide the interface between world datacenter sensors and resources. This approach was chosen because Java ] does not provide such a tight coupling with the underlying operating system. C# ] provides access to the underlying operating system’s functions and thus allows access and real time monitoring of system’s hardware.

## System Output

The most important output provided by the system is the **context repairing sequence of actions**. Having two system management components, one for environment self-healing and one for datacenter self-adapting, there are separate outputs for each component. The repairing sequence of actions is as the name suggests, a sequence of actions which brings the system in an acceptable state from a state in which some policies where broken. The found actions are executed on the corresponding targets.

As secondary output the system stores the <context state, action sequence> pair for future use and creates a Pdf file containing the context monitoring and repair log, for future analysis. An example of a context management log is presented in Listing 5.3. In the presented example three policies where broken (the yellow entries in the log). The resulting repairing action resulted is represented with blue. The initial state in which some policies where broken is represented with red and contains all the environment sensors values, while the states in which no policy is broken are rendered with green.

|  |
| --- |
| Listing 5.3: Context Management Log Example |
| J:\cercetare\licenta\pdf_log.png |

## Architecture Implementation

For implementing the agent-based architecture the Java Agent Development Framework (JADE) ] was used. JADE is a software framework fully implemented in Java [ 52 ] which simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA (Foundation for Intelligent Physical Agents) specifications ]. Another description of Jade would be : JADE is an enabling technology, a middleware for the development and run-time execution of peer-to-peer applications which are based on the agents paradigm and which can seamless work and interoperate both in wired and wireless environment ].

There are several advantages for using JADE. JADE is based on a peer-to-peer architecture so each peer is able to initiate a communication or be subject to a request, providing a fully distributed system. Also, it implements the agent paradigm, applying concepts from artificial intelligence to distributed systems. Agents are active entities, that can refuse a request and which are loosely coupled, thus providing a flexible infrastructure.

JADE supports agent migration. Each instance of the JADE run-time is called a *container.* The set of all containers is called a *platform* and the platform provides a layer that hides the complexity of the underlying hardware. This enables the agents to migrate from one container to the other in real time.



Figure 5.3: Jade Architecture

Figure 5.3 presents the basic Jade architecture. Each Jade instance is called a *container*, since it contains agents [ 47 ]. The set of all Jade containers is called a platform. As represented in the architecture, Jade can run on almost any computing device, from a J2ME [ 53 ] compatible mobile phone to pocket pc’s and desktops. The ability to host a Jade container in any environment brings a great degree of application mobility, every agent being able to migrate from one container to another seemingly. For example, one Jade agent could migrate from a desktop to a user’s mobile phone when the user leaves the room and return to the desktop when the user returns.

Jade provides an asynchronous inter-agent communication mechanism applied in Figure 5.4 for two agents, Ai and A2. The message passing process consists of three steps. In the first step, the source agent wraps the data it wants to send in the proper message format. The second step is the actual sending of the message. The message is placed in the message queue of the destination agent. The third step takes place at a later time, when the destination agent decides to inspect its message queue, extracts the first message contained in it and processes it.

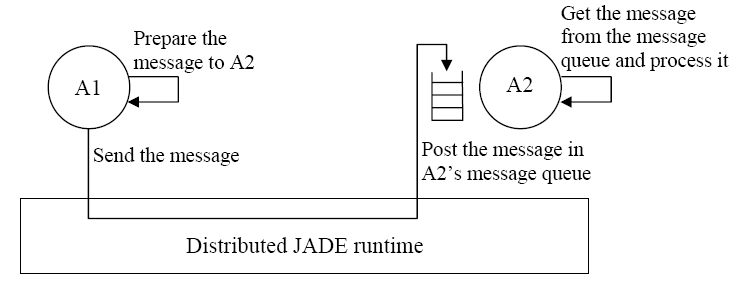


Figure 5.4: Jade Asynchronous Message Passing Mechanism

Jade agents can communicate using various protocols, but for this implementation Java Serialization was used. Another important communication protocol supported by Jade is FIPA SL language [ 50 ] . By using FIPA SL languages two agents communicate using a common ontology as common knowledge. This would have coupled all the agents to the context ontology representation and thus would have reduces the generality of the approach and would have increase the impact a change in the ontology would have had on the system. This is the reason behind using Java Serialization in inter-agent communication.

## Ontology Representation Implementation

### Environment Ontology Implementation

Based on the .owl environment ontology description, a *JenaOWLModel* [ 44 ] is generated and accessed with the specific Jena API. This is further detailed in ] by its author.

### Datacenter Ontology Representation

From the datacenter .owl ontology file description an *OWLModel* [ 40 ] is created using Protégé OWL Loader ]. Using the Protégé ] Ontology Editor tool for generating Protégé-OWL Java code a class structure respecting the ontology hierarchy and containing all the ontology properties is generated. This hierarchy of generated classes has as core the *OWLModel* created from the ontology file. An example of an ontology mapped to java class hierarchy is illustrated in Figure 5.5.



Figure 5.5: Ontology to Java Code Mapping

This is a more useful ontology representation because having Java classes to work with instead of just an underlying ontology model, the ontology concepts can be enhanced with Java specific properties such as *Serializable* or *Cloneable* , providing a better integration of the ontology representation with the rest of the regular Java code. Another major advantage of using Java classes instead of directly accessing the *OWLModel* or *OntModel* [ 45 ] is that all the properties values can be set directly in the ontology entity slot without using the higher *OntModel* functionality. This proved to be an advantage in the experimental results phase when by using a home-brewed policy evaluation mechanism a major improvement in performance was obtained. This improvement was obtained if the properties values were not set also on the *OntModel* and such the Pellet SWRL evaluation mechanism did not trigger at every property change event.

In Figure 5.6 a *CPU* class ontology representation is shown. The *CPU* entity is a subclass of *Component* and has two properties: *associatedCore* and *numberOfCores.* There are cardinality restrictions on the properties: there must be at least one *associatedCore* value associated to each instance of *CPU* and must be exactly one *numberOfCores* value.



Figure 5.6 : CPU Entity Ontology Representation

|  |
| --- |
| Listing 5.4: CPU Entity Java Representation |
|  |

The java code mapping of the *CPU* entity is presented in Listing 5.4. It can be remarked that for each of the properties defined for the *CPU* entity setter and getter methods where generated. Also it can be observer that for the *numberOfCores* property, due to the “exactly one” cardinality restriction, the *getNumberOfCores()* method return *int*, while for the *associatedCore* property which can have more values a *Collection* is returned by the corresponding getter method.

## Agents Implementation

### Context Interpreting Agent (CIA)

The Context Interpreting Agent is responsible for synchronizing the ontology representations with the real context. The agent is implemented by the *CIAgent* class found in contextawaremodel.agents package.

Upon creation, the agent receives from the Context Management Agent described in ] the ontology models for the environment and datacenter. These models are used by the single behavior attached to the agent: *ReceiveMessagesCIABehaviour*. The *ReceiveMessagesCIABehaviour* extends *CyclicBehaviour* and handles the messages received by *CIA*, informing it of newly created instances. For each new instance of *Sensor* and *Server* the behavior adds the instance to the *SensorAPI*.

The **SensorAPI** is based on the API described in ].It implements a pooling data gathering mechanism. At a certain time interval the web service associated to the world element (*Sensor* or *Server)* is accessed and the new data is used to update the ontology representations. This API provides two static methods used for registering a *Server* or a *Sensor* to the pooling mechanism.

|  |
| --- |
| Listing 5.5: SensorAPI functionality |
|  |

The *SensorAPI* functionality shown in Listing 5.5 contains both the mechanism needed for monitoring sensors with discrete values and servers which have a lot of information associated to them. For providing a flexible and low-coupled server information gathering mechanism the Data Transfer Object design pattern [ 37 ] was used. By abiding to this pattern, a *ServerDto* class has been defined with its only purpose to store server information.

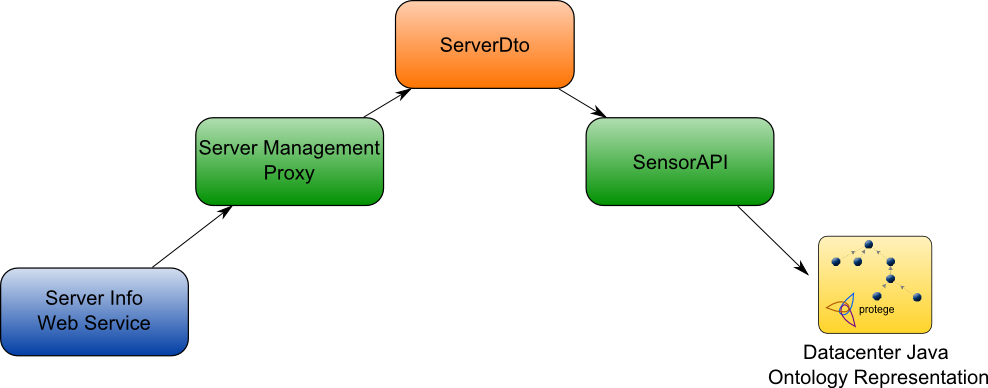


Figure 5.7: DTO Design Pattern SensorAPI Usage

As illustrated in Figure 5.7, the server information is gathered from a web service by a specific *ServerManagementProxy* instance. Depending on the web service type and implicitly on the format of the web service invocation response, the proxy processes the web service response and based on its information creates a *ServerDto* instance. From here on, no matter the source of the server information, only the DTO is transmitted, thus having a uniform mechanism for accessing server information. One the *ServerDto* instance is created, the *SensorAPI* will use it and refresh the ontology data. By employing a object just for data transfer, the particularities of the server information gathering endpoint are hidden from the rest o the application. These particularities are known only by the concrete server management proxy. A change in the server information gathering endpoint would only require an implementation of a different proxy for server management. The rest of the system would be unaffected. This leads to a low coupling between the real world information structure and the internal information representation, providing flexibility to the environment information gathering mechanism.

### Reinforcement Learning Agent (RLA)

This agent represents the core of the infrastructure. It contains the ontology reasoning and solution search processes. The implementation of the agent is available in the *ReinforcementLearningAgent* class located in the *contextawaremodel.agents* package.

Upon initialization, the agent receives from the CMA an *OntModel* together with its underlying Protégé *OWLModel* for both environment and datacenter ontology representations*.* The *OntModel* is used by Pellet ] for evaluating SWRL ] rules, while the *OwlModel* is used in conjunction with a Protégé ] generated *Ontology Factory* for ontology management.

The two main components of this agent, the **Environment Self-Healing Module** and the **Datacenter Self-Adapting Module** are implemented as behaviors extending *TickerBehaviour* . The environment management module is implemented in *contextawaremodel.agents.behaviours* in *ReinforcementLearningEnvironmentManagementBehaviour* and the datacenter management module in *ReinforcementLearningDataCenterManagementBehavior*.

#### ReinforcementLearningEnvironmentManagementBehaviour

This behavior implements the algorithm presented in ] in the context of maintaining the state of the environment within acceptable parameters. The implementation is a joint development effort done in the context of the CONSENS research project ] and is also described in ].

Extending *TickerBehaviour* , the ReinforcementLearningEnvironmentManagementBehaviour implements the *onTick()* method which is called at a specific time interval, interval specified when creating a new instance of the behavior. The *onTick()* method contains the business logic needed for the self-healing system. First the context entropy is computed using . If the computed entropy is different than 0 then the reinforcement action search begins.

For each broken policy, for each broken resource from the policy and for each actuator associated to the policy all the possible actions are simulated and the corresponding states’ rewards are computed. Each action simulation has as effect the creation of a new state which is placed in a list of states, sorted after reward. If no resulting state had the entropy 0, the search continues with the next best state (the state with the highest reward from the states list) and repeats the actions described above. If a sequence of actions which can repair the context exists it will be found and enforced with the help of the *Action Enforcement* *Unit*.

#### ReinforcementLearningDataCenterManagementBehavior

This behavior extends the algorithm presented in ] with enhanced reward computation and elements specific to datacenter management. This behavior is presented in detail in ] by its developer.

#### Logger Unit

This component has the role of logging the context state, the broken policies and the actions taken to repair the context. The log is saved under the form of a PDF file using IText ]. IText is a library used to generate PDF files. The advantage over other libraries is that it supports several mechanisms for PDF creation: java Graphics can be used to “draw” the PDF or specific library classes can be used to create a document structure under the form of paragraphs. An example of a log generated by this component is illustrated in Listing 5.3/

#### Memory Store Unit

This component stores <context state, actions> pairs for future use. When the context is evaluated, if some policy is broken it is checked if the current state hasn’t been encountered before and if it has the action sequence associated to it is executed. Described in detail in ] by its author.

### X3D Agent Implementation

The implementation is available in the X3DAgent class located in the *contextawaremodel.agents* package.

For providing a realistic context representation X3D ] was chosen. X3D is a scalable and open software standard for defining and communicating real-time, interactive 3D content for visual effects and behavioral modeling. X3D provides both the XML-encoding and the Scene Authoring Interface (SAI) to enable both web and non-web applications to incorporate real-time 3D data, presentations and controls into non-3D content. X3D is the successor to the Virtual Reality Modeling Language (VRML). It improves upon VRML with new features, advanced APIs, additional data encoding formats, stricter conformance, and a componentized architecture. The advantage of using X3D is that it provides a simple and easy to use mechanism for defining 3D scene navigation and interaction.

Xj3D ] is a toolkit for VRML97 and X3D content written completely in Java. Although at first Xj3D was based entirely on Java-3D, it has moved from that state and now it provides a faster and less CPU intensive rendering engine. The architecture of the Xj3D is presented in Figure 5.8.



Figure 5.8: Xj3D Architecture

The context representation is described in an .x3d XML file which is loaded by XJ3D and rendered. In the X3D description file every object is wrapped into at least one *Transform* node which specifies the object scale, rotation and translation. Inside the *Transform* node a *Shape* node represents the 3D shape. The *Shape* contains the object’s *Material* and definition. The *TouchSensor* node is used for capturing user input. And finally, everything is wrapped in a *Scene* node.

An example of an X3D node description is presented in Listing 5.6. In the example a *box* is defined and a material called *boxColor*having RGB ACRONIME values 0.5, 0.5, 0.5 (1 representing 255) attached to it. The box is translated on the Y axis by 13 pixels. Also a *TouchSensor* is attached to the box. By clicking on the box a touch event will be generated. The touch event can me caught and processed using an *X3DEventListener* attached to the *BoxTouchSensor*.

|  |
| --- |
| Listing 5.6: X3D File Shape Entry |
|  |

The role of the X3D Agent is to provide a meaningful context representation which can be used for simulating various scenarios or for real-time monitoring of the surrounding environment. More details about the simulated contexts are presented in the Experimental Results chapter.

### GUI Agent Implementation

The implementation is available in the GUIAgent class in the *contextawaremodel.agents* package. The GUI Agent is responsible of the creation and management of the user interface elements used for information output. There are four main user interface components handled by this agent: Environment Monitor, Tasks Monitor, Server Monitors and Logging Window.

#### Environment Monitor

This component is responsible for displaying the environment sensors values, the broken environment policies and the actions taken by the self-healing algorithm. The environment monitor window, represented in , has three main areas: current state area, broken policies area and context repair area.

In the **current state** area the context sensors and their values are displayed. The **broken policies** area displays the name of each broken policy, while the **context repair** area displays the context repair actions. The context repair actions information is displayed in three columns. The first column containing the name of the context repair action. The second column contains the name of the sensor which will be affected by this action, more specific the name of the sensors which has associated the actuator on which this action will be executed. The third and last column contains the value with which the sensor will be influenced. In the effect is the *Decrement* of the value for the *TemperatureSensorI* sensor by 5 degrees.



Figure 5.9: Environment Monitor GUI

#### Tasks Monitor

This component is responsible for displaying the pending tasks queue together with information about the requested resources and is implemented in the *TasksQueueMonitor* class located in the *contextawaremodel.gui.resourceMonitor.taskMonitor* package.

In Figure 5.10 the Task Monitor window is represented together wish several tasks. The monitor window displays the tasks information in two ways under the form of a tab panel. On the first tab just a list with all the tasks monitored by this monitor is displayed (Figure 5.10 left). The second tab displays the entire information available for each task. More exactly, for each task the requested resources ranges and received resources values are displayed. For better visibility, each task receives a randomly assigned color (lighter than the black used for the task names).



Figure 5.10: Task Monitor GUI

#### Server Monitors

This component is the most complex context monitoring user interface module used in this application because it displays information about the running tasks, the total resource usage and the resource usage per task. The component implementation can be found in the *contextawaremodel.gui.resourceMonitor.serverMonitorPlotter.impl* package, the component representing a composition of different classes.

For managing the complexity of the Server Monitor components layout, the components are organized in a tree-like hierarchy, with smaller components being included in larger ones and so on until the entire monitor is created. The architecture of the Server Monitor is presented in .

The Server Monitor has three main components: a Tasks Monitor, a Total Resource Usage Monitor and a Resource Usage Per Task Monitor. The Tasks Monitor consists of the component described at 5.6.4.2 Tasks Monitor. The Total Resources Usage Monitor is a XY Chart plotter which creates a plot of the total resource usage history. The Resource Usage Per Task Monitor creates a Pie chart with a slice for each task running on the monitored server. Each slice represents the amount of resources used by that task. Also, the free resources and resources used by the server’s operating system are shown.

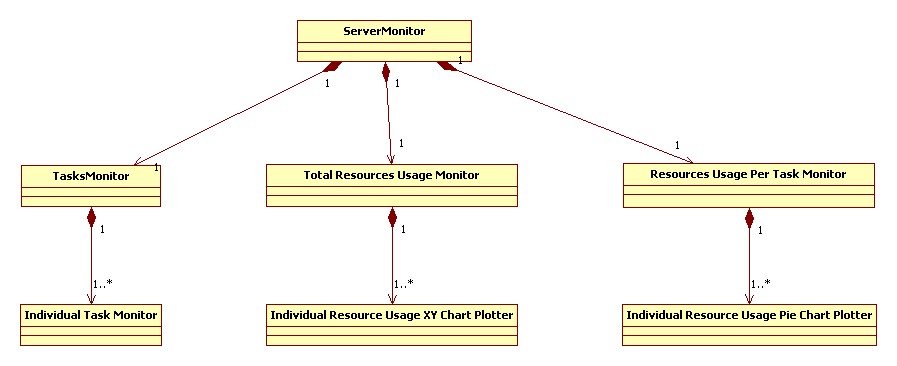


Figure 5.11: Server Monitor Window Architecture

In Figure 5.12 a server monitor snapshot is illustrated. In the snapshot the monitor displays information about a server having two cores and which does not run any virtual machines. In the left of the image is the running tasks display window, in the center is the total resources usage window and in the right hand side is the resource monitor who shows individual resources usage, in this case just the OS and Free.



Figure 5.12: Server Monitor Window

#### Logging Window

The logging window is responsible for displaying periodically the context state, the broken policies and the actions taken for bringing the context in an acceptable state. The component implementation can be found in the *LoggerGUI* class located in the *logger* package.

This component is important because it provides the context management history and makes it easier to trace the system. Two log windows are provided, one for the environment self-healing and one for the datacenter self-adapting systems. An example of a log is shown in Figure 5.12.



Figure 5.13: Datacenter Management Log

## Utility components

### Fuzzy Logic Negotiator

The component is implemented in the *FuzzyLogicNegotiator* class in the *negotiator.impl* package. The Fuzzy Logic Negotiator is the implementation of the negotiation mechanism described in . The implementation of the *FuzzyLogicNegotiator* relies on JFuzyLogic ] , which implements the Fuzzy Control Language(FCL) specification ]. Theis negotiator is used in the self-adapting datacenter scenario for QoS-Energy Consumption tradeoff.

JFuzzyLogic is a package written in Java which provides a mechanism for loading and dynamic management of FCL files. Using JFuzzyLogic the user can define any membership function, modify values or ranges and evaluate the fuzzy rules described in the input file. For deffuzzification JFuzzyLogic implements entirely the FCL specification, supporting Center of Gravity, Centre of Area, Left Most Maximum, Right Most Maximum and Centre of Gravity for Singletons. In Fuzzy Logic, each value has associated a membership value. The membership represents the degree to which the value belongs to the specified range. JFuzzyLogic was chosen for implementing the negotiation mechanism because it makes it easy to define bidimensional functions where one dimension represents the membership of the other dimension to a certain fuzzy value. Also, the provided deffuzzification mechanism implements the computation of the center of gravity in an efficient manner.

In this negotiation mechanism the two negotiating parties are a Server and a Task. The server wants to keep its load inside some optimal parameters, while the task wants more resources than the server is prepared to offer. In this case, the server desirability is decreasing as the load he would need to take increases. The same is for the task. In in the left half the two negotiation parties offers are represented: the server’s with pink and the task’s offer with gray. In this case the desirability function for the server is maximal at 1.500, meaning that the server would be most happy if that would be the result of the negotiation. But the minimum amount requested by the task is 1.650 and it would be happiest to receive 1700. So, the area where the two offers overlap is extracted: 1650 – 1700. Because the task satisfaction is the main priority, the desirability of the task is used for building the negotiation area function, as shown at the right half in . After the new fuzzy function is defined based on the overlapping ranges of both offers and the desirability function of the task, the center of gravity of the resulting area is computed by JFuzzyLogic. The center of gravity reresents the result of the negotiation.

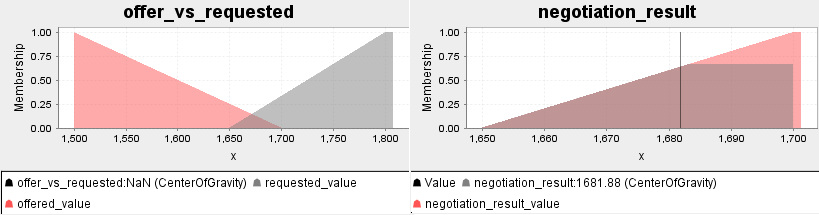


Figure 5.14: Server-Task Negotiation

JFuzzyLogic uses a .fcl file in which the user specifies the variables and their fuzzification and deffuzzification mechanisms. Also, each file has at least one rule block inside which if-then-else conditions are used to assign values to the output variables depending on the input variables. For implementing the negotiation mechanism described above, a (value, desirability) pair is defined for each inflexion point of each offer as shown in . In this example a trapezoidal desirability function is defined, with low and high values being less desirable than values from 1500 to 1700.

|  |
| --- |
| Listing 5.7: FCL specification |
|  |

### Server Management Proxy

For connecting the system described in this book to a real-world datacenter an interface between this system and the datacenter is needed. Also, the interface should have a high degree of flexibility.

In order to support different *ServerManagementProxy* subclasses, needed when porting the system from one datacenter to another, the Factory Method Design Pattern is used to provide a flexible and transparent mechanism for proxy instantiation. The pattern is realized using the *ProxyFactory* class located in the *contextawaremodel.worldInterface.datacenterInterface.proxies.impl* package and is illustrated in .



Figure 5.15: Server Management Interface Architecture

The *ServerManagementProxyInterface* interface specifies the functionality needed for managing a datacenter and represents the contract which every concrete server management proxy must implement, as presented in .

|  |
| --- |
| Listing 5.8: Server Management Interface |
|  |

### Server Information Gathering Endpoint

This component was created to provide real-time server resources usage data. The implementation was developed for servers running Windows Server 2088 R2, using operating system specific functions.

For gathering server data the WIN32 Classes ] provided trough the Windows Management Instrumentation(WMI) ] where used. WMI offers the means to access system information using Win32 classes from any .NET supported language. C# was chosen as the host language due to its support for web services and tight coupling with the underlying operating system. The reason Java wasn’t used for information gathering is that the platform independency of the language makes it hard to query hardware specific information. C# does not have this drawback, being able to access both hardware and software information about the host system. Another advantage of using C# is the easiness with which functionality can be exposed under the form of web services.

The web service created for exposing the functionality of the WMI server information gathering module are hosted on Microsoft IIS Application Server ] and accessed through HTTP ].

|  |
| --- |
| Listing 5.9: WIN32 WMI Specific Functions |
|  |

Chapter 6

# Testing and results

The concepts and algorithm described in Related Work where tested on a simulated smart room, on a simulated datacenter and on real servers.

## Self-Healing Environment Test

This test scenario involves applying the self-healing reinforcement learning algorithm described in to a smart environment. It has as goal to demonstrate the versatility and correctness of the autonomic framework described in this book by using it to manage a smart laboratory.

For this test scenario the autonomic framework is described in this book is used to build a management system for a smart environment having a computer, a camera with face recognition, a light source and an alarm (Figure 2 and Figure 3). Each environment component has attached a sensor for monitoring its state. Also humidity, temperature and room person count sensors have been added to increase the environment’s complexity. Each sensor has a set of possible values encoded using integers for internal representation as described in . There are three policies which the management system has to enforce: *light policy,* *face* *recognition policy* and *temperature and humidity* policy ( ). For interacting with the environment, actuators have been associated to the sensors they have an effect on as shown in .

Table 6.1: Sensors Values

|  |  |  |
| --- | --- | --- |
| Sensor | Possible values | Value encoding |
| Temperature | Z | Any integer |
| Humidity | [0-100]% | [0-100] |
| Light | {ON, OFF} | {1,0} |
| Face Recognition | {Professor, Student, Unknown} | {0,1,2} |
| Computer State | {ON, OFF} | {1,0} |
| Alarm State | {ON, OFF} | {1,0} |
| Room State | {Empty, Not Empty} | {0,1} |

Table 6.2: Policies

|  |  |
| --- | --- |
| Policy | Accepted values combination for policy |
| Temperature And Humidity |  |
| Face Recognition |  |
| Light |  |

Table 6.3: Sensors Associated Actuators

|  |  |  |
| --- | --- | --- |
| Sensor | Actuator | Available actions |
| Temperature | Air Conditioning Unit | {Decrease by 5 , Decrease by 2 } |
| Heater | { Increase by 5 } |
| Humidity | Humidity Controller | {Increase by 3 %,Decrease by 3 %} |
| Light | Light Controller | {Turn ON, Turn OFF} |
| Computer State | Computer Controller | {Turn ON, Turn OFF} |
| Alarm State | Alarm Controller | {Turn ON, Turn OFF} |
| Face Recognition | - | - |
| Room State | - | - |

Two tests have been performed on this scenario. The first test is meant to show the overhead of the learning mechanism used in terms of running time. The second test is an interactive test in which all the possible combinations of broken policies are tested by a human participant using a 3D interactive environment representation.

In order to have a scenario as realistic as possible, C# ASP.NET Web Services where used to simulate the environment sensors. One web service was built for each environment sensor. This approach allows the implementation to be easily switched to use real sensors accessible by web services and thus providing the possibility of testing the system on a real environment. Listing 6.1 presents an example of a simulated sensor: the Alarm sensor. The functionality exposed by the web service for a sensor is always get and set sensor value, allowing the web services to be called uniformly.

|  |
| --- |
| Listing 6.1: Alarm Sensor Simulation Web Service |
|  |

When an action is performed on one of the represented actuators, the actuator influences the sensor values by changing the target web service value according to the specified effect.

### Overhead Test

For the first test in which the context was continuously and randomly broken, the program was run for 28 hours. During this time all sensors received random values as following: for Temperature from 15 to 25, for Humidity from 15 to 35, 0 or 1(“OFF”, “ON”) for Light, Room State, Computer State and Alarm State sensors and 0, 1, 2 (“Professor”, “Student”, “Unknown”) for the Face Recognition Sensor. After the algorithm found a solution the random values assignment was made. The running time of the algorithm is illustrated below in .



Figure 6.1: 28h Random Values Test

Each spike in the plot represents a situation in which an unknown context state has been encountered and a search for the best sequence of actions was performed. In the first 10000 seconds, almost all the running times of the action selection algorithm are larger than 50 seconds. As the knowledge base is being populated with learned knowledge the action search running time decreases, reaching only three running times greater than 50 seconds in the time interval [50000, 70000]. Also an overall reduction in the number and height of the peaks is visible because at each step the algorithm checks if it doesn’t already know the best sequence of actions for the context that it arrived in. If this is not the case, results will be stored in association with the current context. Considering that the number of possible sensor combinations for the chosen ranges is 22.481.940, the self-healing mechanism behaves quite well in rapidly finding and taking the needed actions for fixing the broken context.

### Interactive Test

For the second test an interactive X3D context representation was built using the X3DAgent . The 3D context representation is illustrated in . The representation allows the user to interact with each of the environment actuators by mouse click events. By allowing the user to perform actions on the environment actuators and not on the sensors themselves, the representation can be used to test the self-healing solution on real-world systems.

In this representation there is one actuator associated to each sensor, even for *FaceRecognition* and *RoomState* sensors because in order to fully test the system their values should be changeable to. For altering the environment, the user has to click on the corresponding actuator and the actuator action will be performed. The actuators are represented as follows:

* The *RoomState* sensor has as actuator the green floor. By clicking it the state of the sensor changes to “Room empty: TRUE “or “Room empty: FALSE “. This representation for the actuator was chosen because it can be associated to a pressure sensor.
* The *AirConditioningUnit* is represented in the upper left part of the context screenshot from Figure 6 having the shape of a regular air conditioning unit.
* The *Heater* is represented in the right of the image, under the Alarm.
* The *AlarmController* is represented as a red ball in the right hanbd side of the image.
* For controlling the *ComputerState* the suer must click on the green box representing the computer cenbtral unit, located at the right of the computer screen visible in the left of the image.
* The *LightController* is represented under the form of a white box representing a spotlight in the upper part of the image.
* Finally, the *FaceRecognitionSensor*’s value can be changed by performing a click event on the surveilance camera representation.

For the *AirConditioningUnit* actuator only the *Descrease* by 5 degrees action is available during simulation. For a sensor which has only discrete values, an action performed on its actuator representation will change its state to the next state in the set of possible states associated to it.



Figure 6.2: X3D Smart Environment Representation

Using this simulated environment the output of the system is checked by a human for every possible input scenario. The user modifies the environment such that some policies are broken and then monitors the system’s output sand checks if the suggested sequence of actions is the best for bringing the environment in an acceptable state.

For better understanding, a trace of the environment repair process is presented on the following case: a professor enters in the room, the Computer is “OFF” and the Alarm is “ON”, breaking the Face Recognition Policy. For each resource which doesn’t have an acceptable value for the current policy the learning algorithm simulates executing each action attached to the actuators of the current resource and generates a new context. The first resource which is broken is the Computer, so the algorithm explores every action which brings it to a state different from the current state. Here only one action is possible, so the Computer will be “Set” to “ON” (setting it to “OFF” would not change its state so this action is not considered). For each resulting state we continue with the next sensor that breaks the policy, the Alarm sensor. Similar to the previous sensor, the Alarm can only be set to “OFF”. By performing these actions the Face Recognition Policy is fixed and the context is brought to an acceptable state. This situation is illustrated in the left part of the figure representing the broken context situation while the right hand side of the image contains the monitor described in 5.6.4.1 Environment Monitor, which displays the set of actions which were found to repair the context.

**

Figure 6.3: Face Recognition Policy broken

## Self-Adapting Datacenter Test

For this testing scenario the framework described in this book was used to build an energy-efficient datacenter. This test is meant to further prove the versatility of the solution proposed in this book and to show that self-adaptability and self-healing can be considered isomorphic. The isomorphism between the two self-\* properties can be deduced from the fact that they can be both realized with the same domain model and the same property enforcement protocols.

A second environment ontology representation shown in which contains datacenter specific information has been added and used in the datacenter self-adapting process. The original ontology representation was used for environment self-healing.

The self-adapting extension was tested first on a simulated datacenter and after the solution proved correct on a small home-brewed datacenter containing two servers and a shared storage.

### Simulated Datacenter Test

An X3D representation of a datacenter was built, containing 5 servers and a central control unit arranged in a spoke and arrow layout (). The datacenter has one Temperature and one Humidity sensor. The datacenter cooling system is represented as a giant fan in the middle of the datacenter. The servers are represented as wireframe towers with a solid base. The base color represents the state of the server: gray means *offline* and blue means *online*. The tasks are represented by purple boxes. Each server has a power meter attached which shows the power consumption. For this test the power consumption for a server was considered to be 10\* number of running tasks + 10 Watts.



Figure 6.4: X3D Datacenter Representation

For this test all the tasks are considered having the same priority. Also, the learning algorithm is run for the first time so no action memory exists. For the real case, if the current context has a sequence of actions associated to it in the memory, that sequence of actions is executed or added to the existent path of actions reached so far. For each task, a policy is generated, containing the specified values for the QoS attributes.

The negotiation part described in was not yet plugged in for this test, thus the system was tested on a workload and hardware infrastructure for which at least one deployment strategy which fully respects both QoS and Energy policies exists. Otherwise the algorithm would return the best path it has found and at the next context evaluation time it will search again for ways of repairing the context.

#### Test Datacenter Hardware infrastructure

Concerning the hardware infrastructure, the simulated datacenter contains 5 servers described in the table below (). For each server a policy and associated SWRL rule containing key performance indicators are generated and used in the optimization process.

Table 6.4: Datacenter servers’ characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| Nr | CPU | Memory | Storage |
| 1 | 1 x 3000 MHz | 2048 DDR2-1066 | 250 GB @ 7200 rpm |
| 2 | 2 x 3000 MHz | 2048 DDR2-800 | 250 GB @ 7200 rpm |
| 3 | 4 x 2000 MHz | 4096 DDR3-800 | 146 GB @ 10000 rpm |
| 4 | 4 x 2260 MHz | 6144 DDR3-1333 | 146 GB @ 15000 rpm |
| 5 | 8 x 2000 MHz | 8192 DDR3-1600 | 300 GB @ 15000 rpm |
|  |  |  |  |

#### Test Workload

For simulating a real world scenario we have chosen 5 tasks which are received by the datacenter and are considered to have an infinite life span. The tasks are received in the order they are in the table below ().

Table 6.5: Received tasks SLA requirements

|  |  |  |  |
| --- | --- | --- | --- |
| Task Nr | Task Requirements | | |
| CPU | Memory | Storage |
| 1 | 3000 MHz | 2048 MB | 200 MB |
| 2 | 2 x 1500 MHz | 512 MB | 400 MB |
| 3 | 2 x 2000 MHz | 1024 MB | 256 MB |
| 4 | 8 x 512 Mhz | 240 MB | 128 MB |
| 5 | 3 x 2000 MHz | 4096 | 300 GB |

As initial context the Server 1 is active and the other servers are in a low power state. The basic flow of this example scenario is the following: (i) task 1 is received and is deployed on server 1 which reaches its load threshold and a wake up search which concludes by activating server 2; (ii) task 2 is received and deployed on server 2; (ii) task 3 is received; there is no room to deploy so server 3 is activated; all the task rearrangement and deployment options are analyzed and the solution is to move task 2 on server 3, deploy task 3 on the same server and send server 2 to low power state.

#### Trace

At context evaluation time, the self-adapting algorithm notices that the entropy is larger than zero therefore one or more broken policies exist. The reinforcement learning algorithm will evaluate all possible paths and simulate taking the one with the largest reward. It will reach to a result composed of four actions: deploy task 1 on server 1(), wake up server 3(), deploy task 2 on server 3() and deploy task 3 on server 3(). After this sequence of actions is returned by the reinforcement learning algorithm, the actions are taken in this exact order, bringing the system to an accepted state.



Figure 6.5: Deploy Task 1 on Server 1 Action



Figure 6.6: Wake up Server 3 Action



Figure 6.7: Deploy Task 2 on Server 3 Action



Figure 6.8: Deploy Task 3 on Server 3 Action

After finding a new action plan for a context, the pair (context, action plan) is stored for further reference. The reinforcement learning algorithm will always check first if this situation hasn’t been encountered before and if so, it will take the learned actions.

#### Overhead

After running the algorithm a few times it became clear that the overhead generated by the **Pellet** **OWL 2 Reasoner for Java** ] when evaluating SWRL rules is unacceptable in a real-time application, the rules being evaluated at each change in any ontology value. So SWRL rules where disabled and internal reasoning was implemented using Java. This lead to a decrease in running time of up to 4 times, depending on the type of processor the global loop runs on.

### Real Datacenter Test

The self-adapting extension was also tested on a home-made datacenter running virtual machines.

#### Real Datacenter Hardware infrastructure

The datacenter architecture, as presented in consists of a server cluster (2 servers in this test), a shared storage used by the cluster, environment sensors and a computer running the autonomic system controller built using the framework presented in this book. The shared storage is used to host the virtual machines in order to be accessible from any server in the task migration process. For this there are only two environment sensors, *Temperature* and *Humidity* sensors, which are simulated as described in .



Figure 6.9: Test Datacenter Structure

#### Software Architecture

The software architecture of the test datacenter’s servers is presented in . The servers in the test datacenter are running Microsoft Windows Server 2008 R2 ] as operating system. The hypervisor installed on the servers is Hyper-V Server R2 ]. For this test 2 server where used, each server having a 2 GHz Dual Core processor, 2 GB of RAM and a 150 GB hard drive.



Figure 6.10: Server Technology Mapping

For interacting with the datacenter C# WMI classes where used both for gathering server resources usage data and to access the hyper-v server running on the servers in order to move, import and destroy virtual machines dynamically. The C# classes functionality was exposed trough ASP.NET web services, published on Microsoft IIS Application Server [ 68 ]. The web services are called from within the *Hyper-VManagementProxy* class located in the *contextawaremodel.worldInterface.datacenterInterface.proxies.impl* package and which implements the *ServerManagementProxyInterface*. The Hyper-V management proxy and the WMI Hyper-V interaction mechanism are described in detail in ] by its author. In Figure 6.11 the interaction between the autonomic controller and the server endpoints is shown. A concrete implementation of the *ServerManagementProxyInterface* is used to call the web services publiched on the IIS Server.The web services in turn call the specific functionality on the two C# endpoints, Server Monitor and Hyper-V Manager.



Figure 6.11: Server Interaction

#### Test Workload

For the test, the datacenter workload consists of three virtual machines running Windows XP ] where created using the Hyper-V Management GUI. Due to the lack of time, no dynamic virtual machine resource allocation mechanism was built. Given this fact, the virtual machines physically receive the minimum resources amount specified in when deployed. But inside the control logic, specified ranges are used and each task deployed without negotiation is marked to use the maximum amount of resources required.

For this test the negotiation mechanism presented in 5.7.1 Fuzzy Logic Negotiator was plugged in the controller. The negotiation mechanism was used to negotiate the allocation of resources between tasks when no other solution was possible, after every task reallocation strategy was tested. Such a situation appears when every server in the datacenter is running and can’t accommodate any new tasks while maintaining its optimum load, but could accommodate more tasks if its load would exceed the optimum limits. In this case the negotiation mechanism is used to find a tradeoff between the optimum load and the resources required by the task. After negotiation, the server receives a new optimum load value.

The task requirements were chosen based on the datacenter server’s configuration to use less RAM in order to be accommodated on the servers. Windows Server 2008 R2 together with Hyper-V Server, IIS Server and other programs use around 1GB of RAM of the 2 available on the test servers, so the amount of RAM the virtual machine use was a limitation.

Table 6.6: Virtual Machines Resource Allocation

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Task Nr | Task Requirements | | | | | | |
| CPU (Mhz) | | Memory (MB) | | Storage (GB) | | |
| Min | Max | Min | Max | Min | max | |
| 1 | 100 | 500 | 100 | 300 | 1 | | 2 |
| 2 | 2 x 300 | 2 x 500 | 100 | 300 | 1 | | 2 |
| 3 | 2 x 100 | 2 x 200 | 50 | 400 | 1 | | 2 |

The virtual machines described by Table 6.6 where created using the graphical user interface of the Hyper-V Manager. Each virtual machine has a lightweight Windows XP version installed as operating system. For each virtual machine a java is set to run when the operating system loads, program which performs some computations in an infinite loop, using the virtual machine CPU at 100%.

Using these virtual machines two tests were performed. In the first test, by cloning the virtual machines, workload test sequences were generated randomly. A random number of test sequences of random length scheduled to execute (to be received by the system to be deployed) at random intervals of time where used to test the datacenter management system under real world loads.

The second test type involved testing if the system allocates the resources accordingly and if the virtual machines deployed by the application are usable. This test only involved the three virtual machines described in Table 6.6. In this case all the three machines can be fitted to only one server.

#### Encountered Issues

There were several problems encountered when applying the autonomic system’s management solution to a self-adapting datacenter due to the drawbacks of existing technologies.

##### Server wake up time problem

The server wake up time turned out to be a problem. As the authors of [ 23 ] also encountered for Linux servers, a server running Windows Server 2008 takes between 5 to 10 minutes to start. This means that is not feasible to turn off unused servers because the amount of power they use when starting and the time it takes to start such a server can greatly reduce the response time of the datacenter. Also, there is no currently available mechanism for turning on a Windows Storage Server that was turned off.

The solution to the problem was to send the server in “Sleep” mode. A server in sleep still consumes energy, but only a small amount, but it recovers from sleep in less than 1 minute.

Following the “send to sleep” solution, the server wake up mechanism consists of sending a magic packet containing 16 times the server’s MAC address the datacenter router which forwards it to port 9 of the server. By enabling Wake On LAN from the server network card, the event of receiving the magic packet will wake up the server from the sleep state.

##### Send server to sleep problem

Hyper-V Server disables the Sleep and Hibernate options of the Windows Server before running. Also, the Sleep and Hibernate options can’t be activated once turned off by Hyper-V without a system restart.

For solving this problem, a registry modification was made in order to ban Hyper-V Server from running automatically when the system started. Further, the send *server to low power state* action was implemented in two steps. In the first step the server is restarted and a small program is set to run on startup. When the server restarts, the program set to run on startup sends the server to sleep. This solution is affected by first problem described in 4.1.1.1.1 Server wake up time problem, the time needed for a Windows Server 2008 system to boot. But this would not represent a problem if a load prediction mechanism would be employed that could estimate that the server sent to sleep would not require to be brought online in the next 10 minutes.

##### Dynamic task migration problem

Hyper-V supports real-time task migration with minimum downtime only under special conditions. More exactly, the shared storage used by the server cluster must support Internet Small Computer Interface (ISCSI) [ 72 ] with Persistent Reservation. This reservation is implemented in ISCI-3 standard and allows reserving persistently a storage area for a particular server. Unfortunately, the only Software Storage Servers which implement this functionality and can be included in a Windows Server 208 R2 cluster are servers running Windows Storage Server 2008. Windows Storage server 2008 is available only through Original Equipment Manufacturer licensing, preinstalled with built Storage Servers.

To solve this problem, the task migration was, just as “send server to sleep operation”, performed in two steps: in the first step a snapshot of the virtual machine is exported to the shared storage and in the second step that snapshot is imported onto the desired server by the Hyper-V Server running on it. The drawback of this implementation is that exporting and importing a virtual machine with a hard drive usage of 500 Mb takes up to 4 minutes, delay time observed in the real datacenter test.

##### Virtual Machines Resource Allocation Issue

Even if for a virtual machine it was set that is must receive 50% of one CPU core and the task manager of the windows installed inside the virtual machine showed that the virtual machine CPU is loaded at 100%, Hyper-V reports that the virtual machine uses 2-3% of the total CPU time. This is due to some optimizations done internally by Hyper-V Server and for the time being this issue was left unsolved.

##### Maximum Virtual Machines Number

The Windows Server 2008 R2 version used for the servers in the datacenter was the Enterprise edition, which allows only 4 virtual machines to be run on a server. This means that when our application tried to deploy and start a 5’Th virtual machine the result was an error coming from the operating system.

Chapter 7

# Conclusions

## Results

The solution to building autonomic systems proposed in this book has been built together with Copil Georgiana ]. The solution is a extensible agent-based framework that can be used to create and manage autonomic systems. The framework uses a reinforcement learning based learning mechanism in which the system decides what course of action to take based on the expected reward. The framework also contains the necessary support for context data gathering, data reasoning and action enforcement.

For the reinforcement learning mechanism a means of computing the expected reward based on the context entropy was presented. Also, a context evaluation mechanism based on based policies and weights is presented.

The presented framework was first used to build an autonomic self-healing laboratory. The resulting system was extensively tested both in terms of overhead and correctness. The framework was extended with some domain-specific information in order o be applied on a self-adapting datacenter, where it controlled both the smart environment which housed the datacenter and the datacenter workload distribution. By using this approach to enforce energy policies the power consumption of a datacenter is reduced.

A negotiation mechanism to be used in computing QoS-Energy tradeoffs was presented. The mechanism uses bidimensional functions as negotiation offers, one dimension representing the desirability of the negotiating party to offer the value on the other dimension. The negotiation mechanism was implemented efficiently using a fuzzy logic library and used for the energy-aware datacenter

The approach to datacenter workload representation was the use of virtual machines, which provide dynamic migration from one server to another and give an independent environment for the client to use. Another reason behind the virtual machines approach is that a client can connect remotely to his virtual machine and tailor it after its specific needs, installing or uninstalling software. This gives the client the ability to personalize and tailor the virtual machine while keeping his data isolated from the other clients.

3D context representation where built both for the smart room and the energy efficient datacenter. The smart room 3D representation allowed user interaction and could have been used to monitor and manage a real environment.

The datacenter management system was tested on a homemade datacenter containing two virtual machines servers and one shared storage server. For interacting with the hypervisor the hypervisor API exposed trough WMI was in turn exposed trough ASP.NET web services invoked from the java application endpoint. The servers’ resource utilization was recorded using the WIN32 API exposed trough WMI, in turn as with hyper-v exposed trough ASP.NET web services.

By mapping the same domain model and the same solutions search algorithm on both a self-healing and a self-adapting environment, the two autonomic systems properties can be considered isomorphic and solvable by the same processes.

The presented solution is a general framework which can be used to build any autonomic system just by modifying the domain model ontology representation and adding domain specific policies. This demonstrates that a reinforcement-learning approach is verry versatile because any real world system can be mapped to an environment described by policies and reward functions.

## Future Work

Regarding the energy-aware datacenter, the current implementation lacks the support to create an install an operating system in a virtual machine dynamically by the user. Also, the live migration mechanism should be used for task migration. For this a suitable storage server needs to be found and the Hyper-V functionality needed for real-time task migration discovered and used. Another future aspect is to install the Datacenter edition of Windows Server 2008 R2, which allows an unlimited number of virtual machines per server.

Another improvement over the existing platform is to integrate Open Sim [ 74 ] as means of context representation and interaction. Open Sim simulator gives the possibility to define detailed interactive 3D environments which the user can navigate using a virtual character. Such a representation would provide a good mechanism for monitoring and managing autonomic environments.

Bibliography

[ 1 ] D.A.Menasce, J.O.Kephart: **Autonomic Computing**

[ 2 ] D.M.Chess, J.O.Kephart: **The Vision of Autonomic Computing**

[ 3 ] R.Sterrit, M.Hinchey: **Why Computer-Based Systems Should be Authomatic**

[ 4 ] M.Parashar, S.Hariri: **Autonomic Computing: An Overview**

[ 5 ] Green **Active Management of Energy in IT Service centres:** Performance Computing Center Stuttgart (HLRS), <http://www.hlrs.de/organization/av/isis/research/distributed-systems/games/>

[ 6 ] **VMware Whitepaper**: How VMware Virtualization Right-sizes IT Infrastructure to Reduce Power Consumption, <http://www.vmware.com/files/pdf/WhitePaper_ReducePowerConsumption.pdf>, pages 4

[ 7 ] **EPA** [Data Center**Report to Congress -FINAL 7-25-07**](http://www.federalnewsradio.com/pdfs/EPADatacenterReporttoCongress-August2007.pdf)**,** <http://www.federalnewsradio.com/pdfs/EPADatacenterReporttoCongress-August2007.pdf>, Pages 6

[ 8 ] M. Poniatowski: **Foundations of Green IT: Consolidation, Virtualization, Efficiency, and ROI in the Data Center,** Prentice Hall, 2009, ISBN-10: 0137043759, ISBN-13: 9780137043750, 978-0137043750, pages xiii

[ 9 ] T.Cioara, I.Anghel, I.Salomie, M.Dinsoreanu, C.Georgiana and **M.Daniel**: **A Reinforcement Learning based Self-healing Algorithm for Managing Context Adaptation,** First International Workshop on Communication, Collaboration and Social Networking in Pervasive Computing Environments (PerCol 2010)

[ 10 ] ALA CU AUTONOMIC IN DATACENTERS

[ 11 ] Ubiquitous Computing, <http://en.wikipedia.org/wiki/Ubiquitous_computing>

[ 12 ] S.Russell, P.Norvig : **Artificial Intelligence A Modern Approach**, Second Edition, Prentice Hall,ISBN-10: 0137903952, ISBN-13: 978-0137903955

[ 13 ] P.C.K. Hung, H. Li: **Web Services Discovery based on the Trade-off between Quality and Cost of Service: A Token-based Approach**

[ 14 ] C. Sakama, K. Inoue: **Negotiation by Abduction and Relaxation**

[ 15 ] Abductive Logic Programming, <http://en.wikipedia.org/wiki/Abductive_logic_programming>

[ 16 ] Y.Chevaleyre, U.Endriss, S.Estivie, N.Maudet : **Reaching Envy-free States in Distributed Negotiation Settings**

[ 17 ] Q.B.Vo, L.Padgham: **Searching for Joint Gains in Automated Negotiations Based on Multi-criteria Decision Making Theory**

[ 18 ] K.Fujita, T.Ito, .Klein: **Finding Nash Bargaining Solutions for Multi-issue Negotiations: A Preliminary Result**

[ 19 ]D. Baker, D. Georgakopoulos, M. Nodine and A. Cichocki: **From Events to Awareness**

[ 20 ] P.Hu, J.Indulska, R,Robinson: **An Autonomic Context Management System for Pervasive Computing**

[ 21 ] IEEE 1451, <http://ieee1451.nist.gov/intro.htm>

[ 22 ] SensorML, <http://vast.nsstc.uah.edu/SensorML/>

[ 23 ] R.Das, J.o.Kephart, C.Lefurgy, G.Tesauro, D.W.Levine, H.Chan **: Autonomic Multi-Agent Management of Power and Performance in Data Centers**

[ 24 ] G.Chen, W.He, J.Liu,S.Nath,L.Rigas,L.Xiao,F.Zhao: **Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services**

[ 25 ] D.Kusic,J.Kephart, J.E.Hanson, N.Kandasamy and G.Jiang:**Power and Performance Management of Virtualized Computing Environments Via Lookahead Control**

[ 26 ] M.Amoui, M.Salehie, S.Mirarab, LTahvildari : **Adaptive Action Selection in Autonomic Software Using Reinforcement Learning**

[ 27 ] M.Salehieand,L.Tahvildari: **A weighted voting mechanism for action selection problem in self-adaptive software.**

[ 28 ] I.Salomie, T.Cioara, I.Anghel, M. Dinsoreanu : **RAP – A Basic Context Awareness Model**

[ 29 ] REFERINTA LICENTA GEORGIANA

[ 30 ] S.C.Patrasciuc : **An Agent-Based Context Management Solution (1)**, diploma thesis

[ 31 ] Nash Bargaining Game, <http://en.wikipedia.org/wiki/Nash_bargaining_game>

[ 32 ] Distributed Systems Research Laboratory, <http://dsrl.coned.utcluj.ro/>

[ 33 ] General Responsibility Assignment Software Principles, <http://en.wikipedia.org/wiki/GRASP_(object-oriented_design)>

[ 34 ]Separation of Concerns, <http://en.wikipedia.org/wiki/Separation_of_concerns>

[ 35 ] Dependency Inversion Principle, <http://en.wikipedia.org/wiki/Dependency_inversion_principle>

[ 36 ] Open/Closed principle, <http://en.wikipedia.org/wiki/Open/closed_principle>

[ 37 ] Data Transfer Object (DTO) design pattern, <http://en.wikipedia.org/wiki/Data_transfer_object>

[ 38 ] Protégé, <http://protege.stanford.edu/>

[ 39 ] Protégé Owl Loader, <http://protege.stanford.edu/protege/3.4/docs/api/owl/edu/stanford/smi/protegex/owl/ProtegeOWL.html>

[ 40 ] OWLModel interface, <http://protege-owl.sourceforge.net/javadoc/edu/stanford/smi/protegex/owl/model/OWLModel.html>

[ 41 ] SWRL, <http://www.w3.org/Submission/SWRL/>

[ 42 ] Pellet, <http://clarkparsia.com/pellet/>

[ 43 ] Jena, <http://jena.sourceforge.net/>

[ 44 ] JenaOWLModel, <http://protege.stanford.edu/protege/3.4/docs/api/owl/edu/stanford/smi/protegex/owl/jena/JenaOWLModel.html>

[ 45 ] OntModel, <http://jena.sourceforge.net/javadoc/com/hp/hpl/jena/ontology/OntModel.html>

[ 46 ] JADE**,** <http://jade.tilab.com/>

[ 47 ] JADE A White Paper**,** <http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf>

[ 48 ] Jade Tutorial: Jade programming for beginners, <http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf>

[ 49 ] FIPA, <http://www.fipa.org/>

[ 50 ] FIPA SL Languages, <http://www.fipa.org/specs/fipa00008/SC00008I.html>

[ 51 ] ASP.NET Web Services

[ 52 ] Java, <http://en.wikipedia.org/wiki/Java_(programming_language)>

[ 53 ] Java Mobile Edition, <http://java.sun.com/javame/index.jsp>

[ 54 ] Java Serialization, <http://en.wikipedia.org/wiki/Serialization>

[ 55 ] C#, <http://en.wikipedia.org/wiki/C_Sharp_(programming_language)>

[ 56 ] IText , <http://itextpdf.com/>

[ 57 ] X3D, <http://www.web3d.org/x3d/>

[ 58 ] XJ3D, <http://www.xj3d.org/>

[ 59] HTTP, <http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>

[ 60] Fuzzy Logic, <http://en.wikipedia.org/wiki/Fuzzy_logic>

[ 61 ] JFuzzyLogic, <http://jfuzzylogic.sourceforge.net/html/index.html>

[ 62 ] FCL, <http://www.fuzzytech.com/binaries/ieccd1.pdf>

[ 63 ] Factory Method Design Pattern, <http://en.wikipedia.org/wiki/Factory_method_pattern>

[ 64 ] Windows Server 2008 R2, <http://www.microsoft.com/windowsserver2008/en/us/default.aspx>

[ 65 ] Windows Storage Server 2008 , <http://www.microsoft.com/windowsserver2008/en/us/wss08.aspx>

[ 66 ] Microsoft Hyper-V Server, <http://www.microsoft.com/hyper-v-server/en/us/default.aspx>

[ 67 ] Windows Management Instrumentation, <http://msdn.microsoft.com/en-us/library/aa394582(v=VS.85).aspx>

[ 68 ] Microsoft IIS, <http://www.iis.net/>

[ 69 ] WIN32 WMI Classes, <http://msdn.microsoft.com/en-us/library/aa394084(VS.85).aspx>

[ 70 ] Hyper-V WMI Provider, <http://msdn.microsoft.com/en-us/library/cc136992(VS.85).aspx>

[ 71 ] Microsoft Windows XP, <http://www.microsoft.com/windows/windows-xp/default.aspx>

[ 72 ] Small Computer Interface, <http://en.wikipedia.org/wiki/SCSI>

[ 73 ] Persistent reservation, <http://en.wikipedia.org/wiki/SCSI>

[ 74 ]Open Simulator, <http://opensimulator.org/wiki/Main_Page>