MULTI-AGENT SOFTWARE ARCHITECTURES

TOWARDS THE APPLICATION OF SOFTWARE ARCHITECTURES IN MULTI-AGENT SYSTEMS

By

SALVADOR GARCIA-MARTINEZ

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2007) McMaster University
(Compute Science) Hamilton, Ontario
TITLE: Towards the application of Software Architectures in Multi-Agent Systems.
AUTHOR: Salvador Garcia-Martinez (McMaster University)
SUPERVISOR: Professor Thomas Maibaum
NUMBER OF PAGES: vii, 86

Abstract

Software Architecture is a concept that arose during the last two decades as a consequence of the need for a structured design at an early stage. Software Architecture is defined as a pattern of interconnected components satisfying some structural rule. Software architectures are widely used in many types of systems; Multi-Agent Systems should not be an exception. Multi-Agent Systems have emerged as a design paradigm for large and distributed systems. They are composed of autonomous elements that work together in order to pursue a common goal. They are mostly used in Electronic Commerce, Human-Computer Interfaces, and so on.

In this research, we investigate the state of the art of Software Architectures in the Multi-Agent Systems field, showing that, generally Multi-Agent Systems do no use the software architecture concept properly and, when they do, they do not show specific architectures for Multi-Agent Systems. The approach followed is based on the analysis of six case studies, which are implemented applications that have been published in some of the most important conferences in the area. Additionally we show that, based on the initial design of each case and existing architectural patterns, it is possible to impose a software architecture on each case.

Furthermore, we analyze the way that the term "software architecture" is used in the Multi-Agent Systems literature, showing that, usually, it refers to abstract architectures proposed in standards and frameworks or to an initial design in a system. In addition we clarify related concepts, such as reference architecture, reference models, architectural patterns and design patterns. Finally, we do an exhaustive comparison of the case studies, which aims to highlight commonalities and differences. The objective of this comparison is to analyze if they share a similar architecture that can be reused in more cases and to show how specific properties of Multi-Agent Systems affect in the design of an architecture.

Acknowledgment

This thesis is the result of two years of research during which time many people have supported me. I would like to thank all of these people for their constant motivation and advise.

I want to give special thanks to my supervisor, Dr. Tom Maibaum, who gave me the opportunity of studying under him at McMaster University. I really appreciate his constant support, and all the meetings, discussions and good advice that he gave me. He always believed in my capabilities and supported me regardless of the circumstances.

Aside from my supervisor, I would like to thank the rest of my thesis committee: Dr. Karman Sartipi and Dr. Alan Wassyng, who asked me very good questions, gave me very helpful comments and reviewed my work in detail.

I am also indebted to many teachers from my undergraduate studies, especially Dr. Mauricio Osorio and Dr. Antonio Sanchez, who have continued to motivate me in the field, and provide constant encouragement, recommendation letters and inspiration.

I want to thank to Lic. Fernando Cabrero, who always encouraged me in the pursuit of my goals, and whose advice, personal experience and support, has been an important part of my personal and professional development.

Finally, but no less importantly, I want to thank my family: my parents Emilio Garcia and Rocio Martinez, who believed on me under all circumstances. I have no words to thank them; with my achievements is the best way to show them how grateful I am. My brothers Rocio and Hector Garcia, whose jokes, bizarre stories, constant communication and their visit to Canada were a crucial part of my development and motivation. I also want to thank my brother Emilio Garcia, who constantly provided me his long-distance support and advice.

I wish to thank everyone who helped me, including the McMaster University community, my friends, and so on, all of whom I cannot list. However, I just want to say: THANK YOU.

Sincerely,

Salvador Garcia
Hamilton, Canada.
August, 2007

Contents

List of Illustrations

1. Introduction

1.1. Overview

Today, the software life cycle is very different from the one used during the early of computing days. In the 80s and 90s, software design was purely based on requirement analysis; however, software designers realized that it is not enough. There were many difficulties in essential parts of the software, the size of the new application was getting bigger and the complexity was increasing constantly. As a consequence, new approaches for designing systems at an early stage began to emerge; therefore, designers began to model systems from more abstract levels.

Software architecture is a concept that arises as a consequence of the necessity of a structured design at an early stage. This concept was informally used for a long time; however, it was formally introduced during the 90's. A software architecture is a collection of interconnected components satisfying some structural rules; its importance resides in that it models a system before the design stage and its applicability can be extended to all kinds of paradigms such as the object oriented paradigm, event oriented paradigm or agent-based paradigm.

The agent-oriented paradigm proposes to model an application using autonomous components named agents. When a distributed system is composed of many agents that collaborate in order to pursue a common goal, it is called a Multi-Agent System. The architecture of a Multi-Agent System is modeled based on the combination of the properties that the agent-oriented approach provides and the unique characteristics that each system provides.

1.2. Hypothesis

Defining Software Architecture as a pattern of interconnected components and connectors satisfying some structural rules, our hypothesis is:

> *Multi-Agent Systems designs are developed using the principles of Software Architecture and there are some architectural patterns which are exclusive to Multi-Agent Systems.*

This hypothesis will be tested through the study of the use of the term "software architecture" in Multi-Agent Systems and analyzing six case studies that have been published in the most important conferences in the field. They cover a wide range of applications based on Multi-Agent Systems and they already been implemented.

1.3. Main objectives

This thesis focuses on the analysis of two main areas, Software Architectures and Multi-Agent Systems, analyzing the main concepts and the existing relation between them. In addition, six case studies will be undertaken in order to explore their software architecture, analyzing the way they are designed and comparing their architecture with existing patterns in the literature. Furthermore, the case studies will be compared with the intention of reviewing the main characteristics of each one, comparing their commonalities and differences, and proposing a possible reusability.

1.4. Specific objectives

In a more specific way, the goals of this thesis are to:

- Review the main concepts of software architectures.
- Review the main concepts of Multi-Agent Systems.
- Study the relation between Software Architectures and Multi-Agent Systems.
- Analyze suggested architectures in Multi-Agent Standards and frameworks.
- Analyze the initial design of six case studies, and propose a possible architecture based on existing architectural patterns.
- Compare advantages and disadvantages found.
- Study the possible benefits of applying software architectures in Multi-Agent systems.

1.5. Case Studies

In this work six case studies based on Multi-Agent Systems (MAS) were analyzed: Robot Disassembly Process using a Multi-Agent System, MASACAD, MASEL, Telemedicine for Diabetes, SIMPLE, and Agent Based Simulation Architecture for Evaluating Operational Policies in Transshipping Containers.

All of them are implemented applications that have been published in some of the most important conferences in the area, such as MALCEB (International Symposium on Multi-Agent Systems, Large Complex Systems, and E-Businesses), IAT (International Conference on Agent Technology), CEEMAS (International Workshop of Central and Eastern Europe on Multi-Agent Systems), etc.

They cover some of the most important areas where Multi-Agent Systems are used, such as e-commerce, information retrieval and management, business process, etc. They are applications developed by research groups and none of

them have an official distribution; they are made just for research-academic purposes.

1.6. Constrains

Even though the cases are documented, in some cases, some interactions and descriptions are unclear, and there are not more resources for getting related information. In these cases, some assumptions were made in order to clarify the problem. If this was the case, during the analysis of the each system the necessary assumptions and their respective interpretations are clarified.

In addition, all the cases are experimental applications that do not have official releases; they were developed for research purposes. Moreover, it is not specified if their functionality is correct; in this work, we assume that all of them work properly and we just focus on the design. Finally, the resulting architectures following a pattern are not implemented yet; therefore, it is not possible to measure their correctness at an implementation level.

1.7. Organization

Chapter 2 presents a review of the literature related with Software Architectures; it explores the background, the definitions used in this work and an introduction to architectural patterns. Chapter 3 reviews the literature related with agent-based systems. It analyzes the main agency concepts and gives an introduction to Multi-Agent Systems. In addition, it introduces the most used methodologies and the situations where this paradigm is best suited.

In chapter 4, different standards and frameworks related with Multi-Agent systems are analyzed, and also it explores the relation between them and the proposed architectural concepts. In chapter 5, all patterns related with the case studies are analyzed, and chapter 6 gives an exhaustive description and analysis of all of them. In chapter 7, the results found are analyzed; and finally, in chapter 8 a conclusion of the research done is given.

2. Software Architecture

2.1. Historical Background

During the early days of computing, the software cycle was completely different to the one that is used today. Code was written in machine language and everything was placed directly in the computer's memory. With the introduction of high-level languages, it was possible to develop more sophisticated applications and the use of specific data types and structure as a common practice.

In the late 1960s, programmers noticed that if the structures were built in an optimal way with a good design, the development of the rest of the program was easier. The development of abstract type proposed a new design level that helped to understand modules with similar objectives in an easier way. This development introduced a new understanding of software architecture, specification, language issues, integrity of the results, rules for combining types and information hiding [15].

As a consequence, researchers began to show more interest in the software design. During the 1980s, software design was based on software requirements analysis; however, the focus was moving towards the integration of designs before the developing stage. There were great advances to describe and analyze software systems; there was the introduction of formal description techniques and the emergence of concepts such as consistency and inconsistency [44].

In the 1990s, software designers realized that a development based just on requirements analysis was not enough. There were difficulties in essential parts of the software development, programs were very large and the software reuse began to be a crucial part for application development. Therefore, the concept of *software architecture* emerged. However, even though this concept was not formally introduced, it was used by programmers for a long time [4].

Software Architecture has emerged as an important practice in the software design process; in the next section the most important concepts will be analyzed in detail. In the next one, a compiler, a very well know example, will be introduced; in the next section an intuition related with different types of architecture will be developed. After that, different concepts related with patterns and design will be introduced; finally, the state of the art of software architectures will be discussed.

2.2. What is Software Architecture?

In a traditional way, Software Designers design the components in a detailed level just based on the requirements. Due to the complexity in the software design process, this is not enough; requirements describe too many details and operations for effective engineering. They describe the system and environment in a very general way and sometimes they generate conflicts between designer and stakeholders.

A very effective way to describe in an abstract way how the system is going to be built is through the use of Software Architectures. Through them a system can be standardized and the communication among stakeholders can be more transparent. The main goal when using software architectures is to help at an early decision stage, and to re-use this architecture when possible. In the literature, some definitions from different points of view of software architectures had been proposed; some of the most accepted definitions are:

- "The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the external visible properties of those elements, and the relationships among them" [4]
- "...structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives"[15]
- "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time"[44]

Based on these definitions, in this work, *Software Architecture* is defined as a pattern of interconnected components satisfying some structural rules. In this definition, a *component* is a unit of structure that encapsulates a set of services and a *connector* is a unit of structure defining an interaction protocol for the components. Examples of components are clients, servers, filters, layers and databases; examples of connectors are procedure call, event broadcast, database protocols and pipes [53]. Among others, software architectures provides the following advantages[4]:

- **Stakeholder Communication**. An architecture has different stakeholders (costumers, users, project manager, etc) that are concerned about different goals in a system. Software Architectures provide a common language that represents their concerns; therefore, it is for all concerned easier to understand large systems.
- **Early Design Decisions**. In a system, the most difficult parts to correct or modify are the early decisions. They have a strong influence on the future effects and are the basis for later development. Software architectures

define implementation constraints, dictate an organizational structure and help to manage easier changes.

- **Re-usable Models**. Re-use on an architectural level provides good solutions for similar systems; therefore, all the properties are just transferred to another system. In addition, systems can be built using already developed elements; an easy use of design alternatives can be promoted.

2.3. Example: Multi-Phase Compiler

An architecture can be described from different points of view. Therefore, they can be constructed following different styles. One of the most popular examples that has achieved acceptance is the multi-phase compiler which will be analyzed through this section. A compiler has five phases:

- *Lexical analysis*. Takes the characters from the source and produce tokens for the next phase.
- *Syntactic analysis*. Using the tokens, it forms phrases that can be either definition phrases or use phrases.
- *Semantic analysis*. It correlates the phrases with elements that are associated with specific definitions; as result it produces correlated phrases.
- *Optimization*. Produces key phrases for the generation of the object code. This phase is optional.
- *Code Generation*. Where code is produced.

A multi-phase compiler can be designed in different ways depending on its goals. The most common one is sequential compiler, where each phase transmits data to the next one in a linear way. Another way to design it is in a parallel style. This is used when one of the mail goals is to optimize processing time. Although both designs have different goals, they are composed by the same architectural elements:

- *Processing Elements*: Lexer, parser, semantor, optimizer, code generator.
- *Data elements*: Characters, tokens, correlated phrases, annotated phrases, and object code.
- *Application Level Properties*: has-all-tokens, has-all-phrases, has-all-correlated-phrases, has-all-optimization-annotations.

2.3.1. Sequential Architecture

A classical Multi-Phase Compiler [44] is based on a sequential architecture, where each phase performs a task before the next phase begins; in this case the data elements are sent in a direct way through the connecting elements, which are composed by procedure call and parameters.

Figure 1 represents the *processing view* of the architecture [44]. It shows the flow of the data through the compiler; Figure 2 shows the architecture from the *data view* [44]. This is captured by the notion of *application-oriented properties*, which describes the most important states of a data structure.

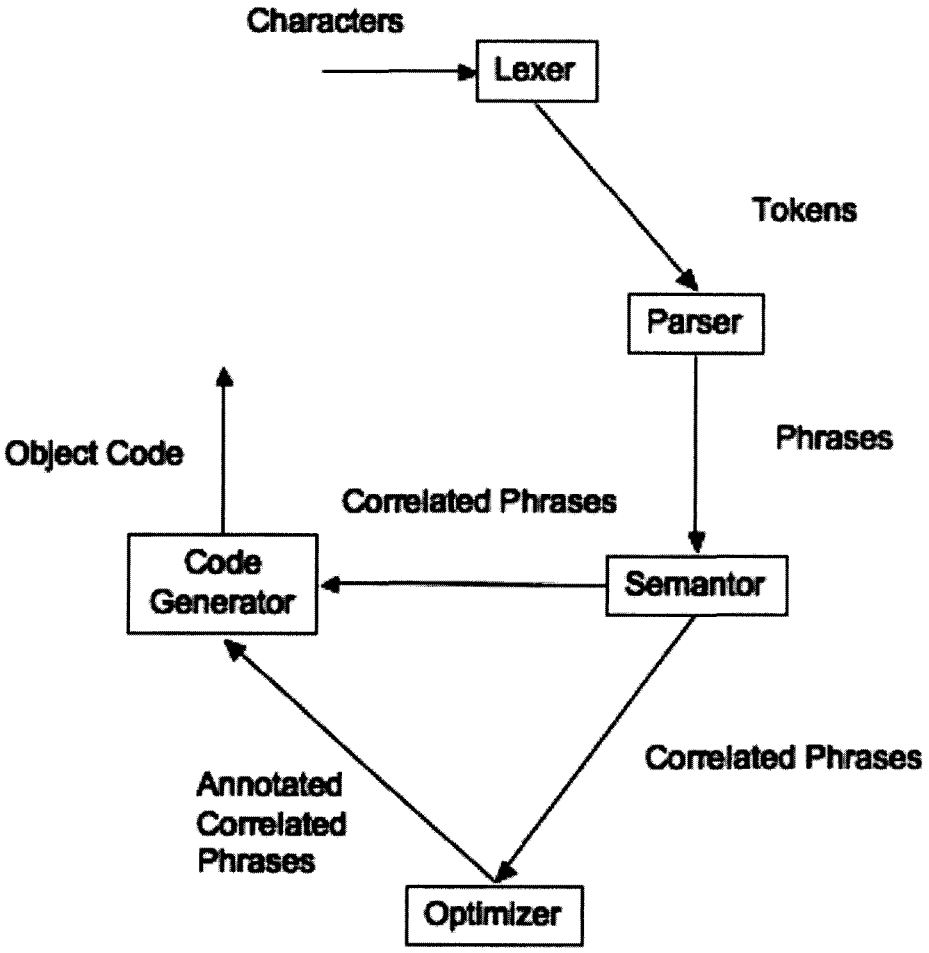

Figure 1. Processing View of Sequential Compiler Architecture

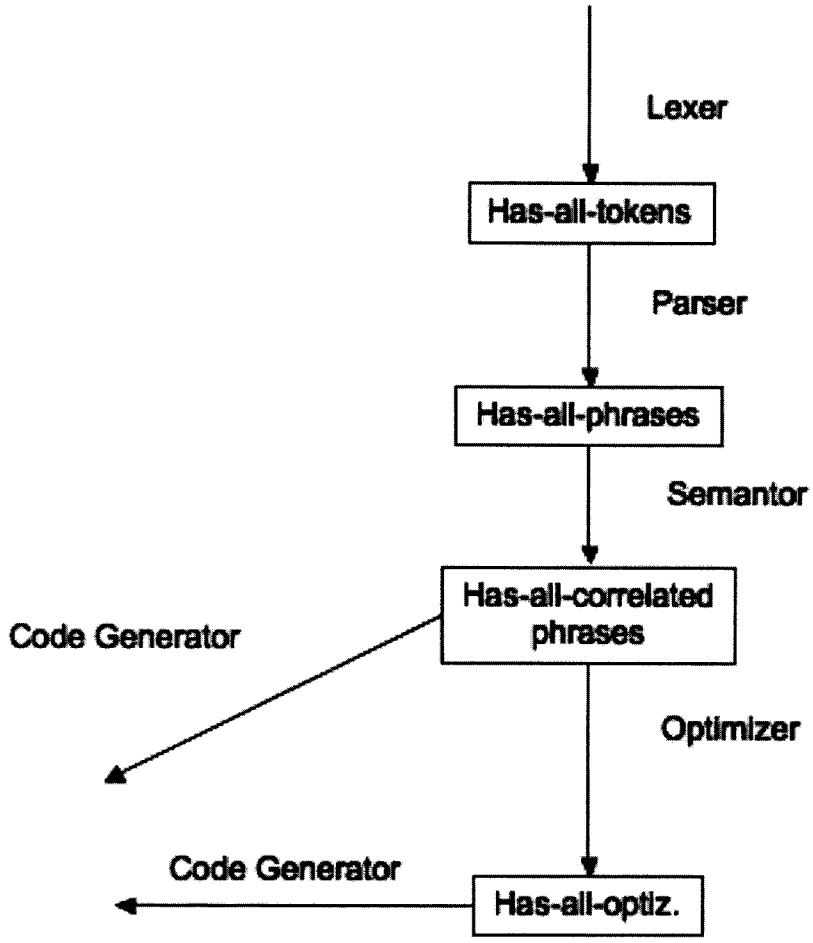

Figure 2. Data View of Sequential Compiler Architecture

In this example the application oriented properties are [44]:

- *has-all-tokens.* Result of the lexical analysis of the code; necessary for the parser.
- *has-all-phrases.* State produced by the parser; necessary for the semantor.
- *has-all-correlated-phrases.* Produced by the semantor; necessary for the optimizer and code generator.
- *has-all-optimization-annotations.* Produced by the optimizer; used by the code generator to begin processing.

2.3.2. Parallel Architecture

When performance is an important factor in the compiler and the execution time must be optimized, a possible solution is to adopt an architecture that treats components in a parallel way. In Figure 3 a parallel version of the

compiler is presented [44]; in this case just the lexer, the parser and the semantor are represented.

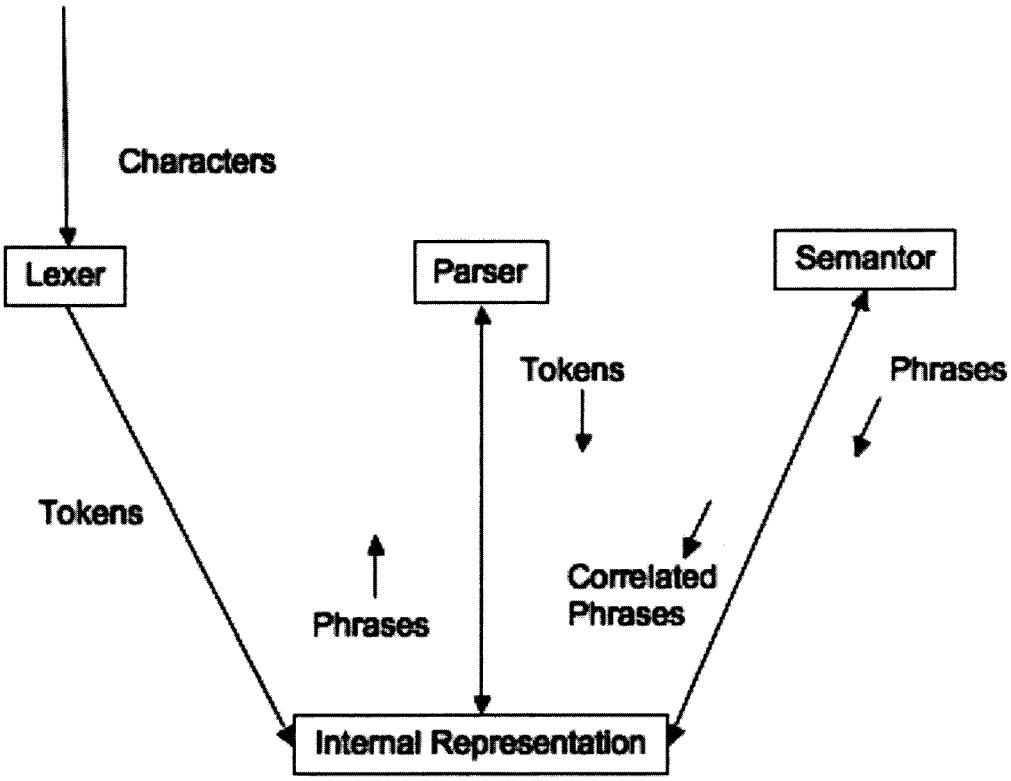

Figure 3. Parallel Process Architecture

As in the sequential case, the notion of application-orientation is used; moreover, they are more complex and they are used for providing coordination and synchronization. The basic application-oriented properties, which describe the states of data structure that are important to the processing elements, are [44]:

- *no-tokens*
- *has-token*
- *will-be-no-more-tokens*
- *no-phrase*
- *will-be-no-more-phrases*
- *no-correlated-phrases*
- *have-correlated-phrases*
- *all-phrases-correlated*

The most important points in this architecture are that the processing elements are very similar to those in the sequential architecture, just with different

use of the properties, and that this architecture can be compared with the previous one. There are related points of processing, which promotes a better understanding.

2.4. Analogies with different architectures

Software Architectures can be compared with architectures from different fields; through this analysis, it is possible to develop an intuition through existing disciplines. The development of this intuition is realized through three disciplines: Hardware Architecture, Network Architecture and Building Architecture [44].

- **Hardware Architecture.** They emphasize the configuration of hardware elements such as CPUs, memory, hard disks, and peripheral devices. Some examples of hardware architectures are RISC, pipelined and multi-processor machines. Two important things that can be compared with software architectures are that in hardware architectures there are a small number of design elements, such as clients, hubs and servers, and scale is achieved by their repetition.

  In software architecture, there are many design elements and the scale is achieved through the addition of new ones. However, there is an important similarity: software architectures are usually organized in an analogous way to the common hardware architectures.
- **Network Architecture**. They abstract the design elements (node and connections) of multiple computers connected through a network.. Some examples of computer networks are bus networks, star networks and ring networks. Two points related with software architectures are important: few components and few topologies. In software architecture, there are a large number of topologies, and it is possible to consider how the components are organized, for example, in a distributed way.
- **Building Architecture**. Even though classical architectures are not very related with the field, there are several points that can be considered for software architectures:
  - *Multiple Views.* Depending on the aspects that need to be emphasized, a building design plan has different *views* such as wiring, plumbing and heating. In an analogous way, software architectures have different views, for example, implementation, communication, interaction between the components, etc. The first one is the most important; it is the detailed view for the architect, like a building without "skin" that shows all the internal details.
  - *Architectural Style.* It classifies architectures according to the form of a building, techniques used to build it or the materials used. From a descriptive point of view, it defines how the design elements are organized and codified, and the relationships between them. Analogously, this is a concept widely found to be one of the most useful in software architectures.

- *Style and Engineering.* This is the relation between engineering principles and architectural styles. In classical architecture, some aspects, such as the position and place of the structures, are not just for aesthetic purposes, they are also functional. In a similar way, in software architecture the engineering behind the design principles is fundamental.
- *Style and Materials*. From a classical point of view, materials have certain characteristics that provide a unique style and different properties to the building. Analogously, in software architecture the properties of distinct components provide different characteristics to the architecture. This relationship is one of the most important parts of software architectures.

2.5. Architectural Patterns, Design Patterns, Reference Model and Reference Architecture

In the context of software design it is usual to find that several problems share a similar solution that can be reused just recurring to the way that the main components were design. This reusable solution is a *pattern.* A pattern is used in two ways: Architectural and Design Patterns. Even though both are very related, their purpose and concepts are very different.

A system is composed by a set of components that are interconnected. An *architectural pattern,* also named *architectural style,* is the structure of these components and the way that they are related. It specifies their responsibilities and the guidelines to organize them [9, 10, 15]. Some examples of possible architectural patterns are Layers, Pipes and Filters, Blackboard, Broker, Model-View-Controller, Presentation-Abstraction-Control, Microkernel and Reflection Pattern [10].

A design pattern refines the design and relation of the components that were specified in the architecture. It specifies their responsibilities and the way that they are communicated in a more detailed level, for example, they specify the classes of a system and the way that they are related. A software architecture specifies the structure of a system with a higher level of abstraction. Examples of design patterns are Whole-Part, Master-Slave, Proxy, Command Processor, View Handler, Forwarder-Receiver, Client-Dispatcher-Server and Publisher-Subscriber pattern [10].

Two more concepts that are very related with software architecture are reference model and reference architecture. A *reference model* divides the problem into parts depending on the functionality. It standardizes a problem (usually situated in a mature field) into parts that together solve the problem. A compiler is a typical example of a reference model; there are different kinds of

compilers, but all of them share the same parts: a lexical analyzer, syntax tree, symbol table, semantic analyzer and code generator [33, 54].

A *Reference Architecture* is based on a reference model, but maps it onto software elements that implements the functionality described in the reference model. It transforms vague notions into concrete implementations full of details [48, 38].

All of these concepts are closely related, however, they are not software architectures; they are just concepts that help to capture the elements of an architecture. The relation between them is presented in the (Figure 4). The arrows indicate that the next concepts contain more designs elements. Reference Model is enclosed in a circle because it really does not represent a direct relation with the concept of architecture (components and connectors); the rest of the components are represented as a box because they explicitly use components and connectors [9].

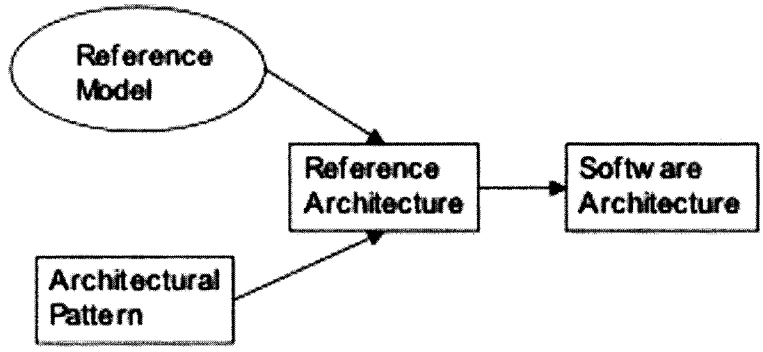

Figure 4. Relationships between Reference Model, Architectural Pattern, Reference Architecture and Software Architecture

2.6. Present and Future of Software Architectures

Software Architecture has already been through several phases, such as Basic Research, Concept Formulation, Development and Extension, and Internal Enhancement and exploration. Currently the development is in a external enhancement and exploration, and popularization phase [34].

During the external enhancement and exploration phase, several areas had reached enough maturity in order to be useful outside of research and development groups. One example is the Unified Modeling Language (UML) that currently is integrating a considerable number of new design concepts related

with Software Architectures. In addition, the work on general Architecture Description Languages (ADLs) that are used for describing systems and software architectures, are increasing constantly and adapted to specific architectures.

Architectural Patterns, specific purpose architectures such as service-oriented and agent-oriented architectures, and different standardizations reflect the popularization of Software Architectures. In addition, today they are not just concepts that are taught on a graduate level, now they have been introduced in a undergraduate level; even in industry, the job title "software architect" is gaining popularity. The current status reflects to the fact that now Software Architecture is beginning to be considered as a true engineering discipline.

Trough the integration of Software Architecture into the design process, several mechanisms are clearer; therefore, software development is more powerful. Now, researchers have the responsibility to show that their now ideas are promising and effective and keep doing research in innovative areas. Some of the areas that should be developed are:

- Finding a proper language for representing architectures.
- Finding ways to assure the relation between architecture and code.
- Re-designing software testing techniques using software architecture concepts.
- Organizing and standardizing concepts in order to create reference materials.
- Support for adapting changes in resources related with user's expectations and preferences.

3. Multi-Agent Systems

3.1. Introduction to the Agent-Oriented Paradigm

Through software design history, several engineering paradigms, such as procedural programming, structured programming, object-oriented programming, etc., have emerged. All of them claim to be the best option in order to make the engineering process easier; however, researchers are continuously looking for a more powerful techniques satisfying more demanding applications.

A system can be designed using different techniques; for example, if modularity is a main concern, the object-oriented paradigm is a good option. If a module is based on a specific algorithm, imperative programming is the best possibility. In a similar way, when the key feature in a system is autonomy, i.e. to reach a desired goal without the user's interaction; a design possibility is the *agent-oriented paradigm.*

When a system is large and complex, software engineers have to develop several mechanisms for addressing it. Three of the most used are [28]:

- **Decomposition**: When a system is very large it is very useful to divide it into smaller parts that can be treated in an isolated way. Therefore, the designer can focus his attention on a particular part of the problem.
- **Abstraction**: This technique simplifies the problem emphasizing some details and properties, and hiding others. Through this technique, the designer can focus his attention on the most important aspects of the problem, leaving many details for the next stage.
- **Organization**. The identification of relations between the components of the system. Basic components can be grouped at a higher-level and be treated in the same way by the parent, i.e., in a hierarchical way. In addition, several components can work together sharing similar functionalities.

Every emerging paradigm has to manage these strategies; the agent-oriented paradigm integrates them in a very effective way. However, it does not intend to change the perception of programming; it is a tool that helps with a particular set of problems. In the Agent Oriented Paradigm, the unit element is named *agent.*

An agent is a component of the system that, with the help of other agents, can solve a problem in an autonomous way. During the past years, there have been different points of view and discussions about how an agent can be defined; two of the most accepted are:

- *Shoham*: "An entity whose state is viewed as consisting of mental

components such as beliefs, capabilities, choices and commitments" [45]. In this definition an agent is viewed as a component capable of reacting according the changes that surround it (beliefs), and depending on that, it is capable of choosing the best options to pursue a goal.

- *Wooldridge*: "An encapsulated computer system situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives" [59]. Similar to the previous definition, this one suggests a component with capabilities to choose an action (depending on the environment) to meet its objectives.

Depending on the situation, *autonomy* can be defined in different ways. In this case, it refers to the idea that "agents are able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behavior" [60]. Some examples of agents are:

- **Any Control System**. A thermostat is a simple agent example. It has a sensor that detects the room temperature; if the environment temperature is too low, it will turn on until reaching the desired level. Once the temperature is ok, the heating will turn off. This system is automatic, it does not need the interaction of a user for reaching its desired objective, and therefore, it can be considered as an agent.
- **Software Daemons**. Programs that run in the background without control of the user. They are initiated through processes. For example, in X-Windows there is the *xbiff* utility. It checks continuously for incoming emails; when it receives one, the system notifies it to the user.

3.2. Intelligent Agents

An agent, by definition, is an autonomous process, but, when intelligence capabilities are added, it can decide which actions to choose in order to pursue its goals, and how to interact and communicate with the other components. Wooldridge defines an Intelligent Agent as "an agent that is capable of flexible autonomous action in order to meet its design objectives; where flexible means three things: reactivity, pro-activity and sociability" [60]. In this definition, there are three concepts worth analyzing:

- *Reactivity* means that the agent can react to the changes occurring in the environment or to the decisions made by other agents.
- *Pro-activity* connotes that the agents can decide which actions they are going to take in order to satisfy them objectives.
- *Sociability* expresses the capability of an agent to interact with the others in order to satisfy its goals.

An agent architecture is a structural model of the components that constitute an agent as well as the interconnections of these components. Together they compose a computational model that implements the basic capabilities of the

agent. Intelligent agents can be formalized from an abstract architecture point of view, or like a concrete architecture [57]. Due to the scope of this work, internal architectures will not be analyzed; however, a brief description of them will be given.

Abstract architectures are mainly general agent models that do not get into detail about how they are going to be implemented. The most important ones are:

- **Purely Reactive Agents**. Agents that take decision without considering the history; they simply respond to the environment.
- **Perceptive Agents**. Agents that take decisions through two subsystems: perception and action. The first one captures observations from the environment; the second one represents the decision making process.
- **Agents with State**. They have an internal structure where the history and the environment state are saved. The decisions are mainly based on the information acquired from the internal structure.

In contrast with abstract architectures, concrete architectures are more specific about internal structures and they focus on the operation of agents. They can be of four types:

- **Logic-Based Architectures**. They make decisions based on logical deduction. Agents are viewed as theorem provers and their behaviors are specified in a formal way. This specification is refined through several stages until an implementation is reached.
- **Reactive Architectures**. Decisions are implemented based on a mapping from situation to actions. Agents react to an environment without reasoning; their decisions are realized through a set of behaviors that can be fired simultaneously.
- **BDI agents**. They are based un the philosophical concepts of *practical reasoning*, where the decisions depend on the manipulation of data structures that represent the beliefs, desires, and intentions of the agent. They can decide what goals they want to achieve and how they are going to be achieved.
- **Layered architectures**. In this case, different subsystems can be arranged in a hierarchical way. The decisions are made via software layers that reason about the environment at different levels of abstraction.

3.3. Multi-Agent Systems

When different intelligent agents collaborate in order to pursue a main goal, they belong to a society that is named *Multi-Agent System*. Durfee and Lesser define a Multi-Agent System as a "loosely coupled network of problem solvers that work together to solve problems that are beyond the individual

capabilities or knowledge of each problem solver" [27]; therefore, a Multi-Agent System is a distributed and asynchronous set of agents that cannot solve problems by themselves; they need to cooperate with the others to pursue a common goal.

A multi-agent system is used when a problem is too large for a centralized system. It allows the communication, interoperability and interconnection between heterogeneous systems, providing a distributed solution. It also suggests an infrastructure that enables the interaction between agents at a social level. Finally, it provides middleware that supports the communication and coordination of activities [42].

The system can be designed in a centralized and in a distributed way. Within a centralized approach, the system can be efficient, but sometimes, when the information is not available at the same location, it is very hard to centralize the system. With distributed systems it is possible to cover different domains, locations, and distribute the tasks. They are easy to understand, and also they provide an infrastructure for specifying communication and interaction protocols. In addition, they provide a set of agents that are autonomous, distributive and cooperative [57].

Multi-Agent Systems are a particular type of distributed intelligent system that are is by many autonomous agents that can interact which each other. They are placed in an environment surrounded by knowledge that influences the actions that are based on decisions and goals. They exchange knowledge and negotiate the necessary information in order to pursue a common goal.

In Multi-Agent Systems, agents work within an environment that provides the infrastructure for the interaction and communication between agents. Communication protocols enable agents to exchange and understand messages; interaction protocols enable agents to have a structured exchange of messages through conversations.

3.3.1. Communication Protocols

In order to better achieve their goals, agents must communicate in order to interact successfully; for that, they must satisfy a set of properties [22]:

- *Communication.* Enable agents to coordinate their actions and behavior in a coordinated way.
- *Coordination.* Allow agents to perform different activities in a shared environment.
- *Cooperation.* Coordinates non-self interested agents.
- *Negotiation.* Establishes the communications between competitive or self-interested agents.

- *Coherence*. The way that all the agents behave together as an unit

All the agents within a system have communication given by the combination of syntax (symbols), semantics (what symbols denote) and pragmatics (symbol's interpretation); this communication is possible through messages that can be *assertions* or *queries*. In addition, the communication is given by protocols that can be specified at different levels such as:

- **Lowest level**. Specifies the method of interconnection.
- **Middle Level**. Specifies the format and syntax of the information.
- **Top Level**. Specifies the meaning and semantics of the information.

Moreover, the communication protocols can be binary or n-ary and they are specified by structures composed by the following fields:

- *Sender*
- *Receiver*
- *Language in the protocol*
- *Encoding and decoding functions*
- *Actions to be taken by the receivers*

The information and knowledge exchange is specified by the protocol KQML (Knowledge Query and Manipulation Language). In order to have an understandable communication between agents, an *ontology* is created. An ontology is a common vocabulary of agreed upon definitions and relationships between those definitions, to describe a particular subject domain. KQML has an infrastructure that is not part of the specification. It allows the agents to locate each other; this function is performed by routers and facilitators.

Agents need a description of the environment where they are acting. For that they use the knowledge that exists within the environment, which is expressed through a *knowledge interchange format*. The most common, also used within expert systems, databases, intelligent agents, etc. is KIF. It is a logic language that has the main characteristics of First-Order Logic and mediates the translation between different languages.

3.3.2. Agent Interaction Protocols

Interaction protocols govern the exchange of messages among agents. In cases where the agents have conflicting goals, the objective of the protocols is to maximize the use of the agents. In cases where the agents have similar goals, the objective is to maintain globally coherent performance of the agents without violating their autonomy.

The actions of multiple agents need to be coordinated in order to manage the dependencies between agents' actions. There is a need to meet global constraints, and no one agent has sufficient competence, resources or information to achieve system goals. Interaction protocols realize this coordination; they can be of five types: Coordination Protocols, Cooperation Protocols, Contract Net, Blackboard Systems and Societies of agents.

- **Coordination Protocols**. They assist the agents in order to coordinate their activities satisfying their interest or the goals of the group. They also coordinate the dependencies between actions in order to meet global constrains.
- **Cooperation Protocols.** They decompose the distributed task. The system can choose among different decompositions considering the resources and capabilities of the agents.
- **Contract Net.** It is an interaction protocol for cooperative problem solving among agents. It provides a solution in order to find an appropriate agent to work on a given task.
- **Blackboard systems**. Everybody posts messages, and there is a control loop that executes the systems until a decision is taken.
- **Societies of agents**. Intelligent agents do not function in isolation, they are at the very least a part of the environment in which they operate, and the environment typically contains other such intelligent systems. Social commitments are the commitments of an agent to another agent. These must be carefully distinguished from internal commitments [22].

3.4. Methodologies

As part of agent-based systems development, different methodologies have been developed. A methodology is "the set of guidelines for covering the whole lifecycle of system development both technically and managerially" [8]. A methodology is intended to help understanding of a system, and as a result, for designing it. Usually, a methodology begins with a model that has a high level of abstraction; as the analysis continues, they become more concrete and detailed, i.e., they become closer to the implementation.

Methodologies help to identify the task of the system, to identify which components can be represented as agents and to define their interactions and protocols. In addition, they also define the interaction between the agents, the environment and their behavior. They help in the software engineering process providing different tools, notations and languages that enhance the process for the designers. Three of the most known methodologies are: Gaia, Tropos and MaSE.

- **Gaia Methodology** [30]. It is mainly based on five models: role, interaction, agent, services and acquaintance models. It uses standard

notations like UML and AUML. It also has two phases: analysis and design. The first one defines the environmental model, and the second defines the organizational rules and structures.

- **Tropos Methodology** [46]. It is founded on the relation of an agent and its mentalistic notions in an early requirements analysis. Possible requirements capture is based on the *i* model* [62] that specifies how the system is going to be. The development is based on four phases: early requirements analysis, late requirement analysis, architectural design and detailed design.
- **MaSE Methodology** [13]. This methodology is more focused on heterogenous Multi-Agent Systems. It is a derivation of UML and its objective is to assist the designer in order to get an initial set of requirements. It has two different phases:
  - *Analysis phase*. Produce roles and tasks and analyzes the goals. Goals are what the system is trying to achieve. The roles perform some function for the system. In this phase, the goals are captured, use cases are applied and the roles are refined.
  - *Design Phase*. Creates agent classes, constructs conversations, assembles agents and develops the software design.

Current methodologies try to adapt object-oriented approaches; however, they have the disadvantage that this decomposition is different from the one in the agent-oriented paradigm. Good methodologies should encourage the designers to analyze the system based on agent concepts, not objects. In addition, with the object oriented methodologies, some agency properties, such as proactivity and dynamism are hard to analyze. It is difficult to compare objects and agents; however, one of the main goals in the agent-based paradigm is to reuse and extend the concept of object in an agency terminology.

3.5. Multi-Agent Applications.

Using agents can be very helpful in different fields and systems. Michael Wooldridge proposes the following main application fields where agents are mostly used [61]:

- **Agents for Workflow and Business Process Management.** The concept of *Business Process* is defined as "a set of logically related tasks performed to achieve a defined business outcome" [17]. Workflow systems try to automate processes of a business; in this case Business Process Management is the intersection between Informatics and the Business Process.

  ADEPT [39] (Advanced Decision Environment for Process Tasks) is a project that presents an approach to implement systems for managing the business process. It consists of several agencies organized in a hierarchical way that provide services to negotiate agreements in the system. The

resulting Systems relate different sections of the organization and define a structure using all the concepts related with agents.

- **Agents for Distributed Sensing.** Multi-Agent Systems are very helpful when an application is composed of a network of sensors that must cooperate in order to get or sense all the information. An example is a battlefield where the sensor should track all the vehicles that pass in a determined range. Sensors can exchange predicted information with the others when a vehicle passes from one region to another.

- **Agents for Information Retrieval and Management**. When in a system there are distributed information sources, agents are a very good approach. They must interact with different sources and manipulate the information; these sources can be databases or more information agents. Some applications in this category are *web agents, personal information agents* and *retrieval systems*.

- **Agents for Electronic Commerce**. This is one of the main fields where agents are used. They are promoted in order to help the consumer to find the best deal during the six main stages in e-commerce: Need identification, product brokering, merchant brokering, negotiation, purchase and delivery and product service and evaluation. Some of the most usual agents in this field are comparison-shopping agents and auction bots.

- **Agents for Human-Computer Interfaces**. Usually users interact with systems in a direct way, however, sometimes it is useful if the interface can take decisions by itself. Multi-Agent Systems are very helpful in these cases, as they can help the user in an adaptive way to take their own decisions; these types of agents are known as expert assistants or interface agents.

- **Agents for Virtual Environments**. Virtual Environments are used to simulate an artificial world; they are usually used in the cinema and videogames. Agents help to simulate this world in a realistic way; they need to show emotions and empathy with the understanding of human behavior.

- **Agents for Social Simulations**. Agents can be used to simulate human behaviors in a society. Each agent can represent an individual person and together will represent a team or an organization. A good example is the simulation of a soccer game using robots.

- **Other Agents**. In addition, agents have been proposed for many others areas such as industrial systems management, where they are very useful during the industrial process. They are also used for spacecraft control and for air-traffic control, where they assist the coordination and control of the vehicles and different tasks, depending on the respective case.

3.6. Software Architectures and Multi-Agent Systems

A software architecture is designed as the result of requirements modeling in early stages, the chosen paradigm, and the specific properties of a particular system. In case of Multi-Agent Systems, they present properties such as distributivity and cooperativity that restricts the way in which the architecture is structured.

In addition, the internal architecture of an agent influences the way that components communicate within a system. For example, reactive agents add event-based properties to the system; therefore, the chosen architecture can be designed using an implicit communication style.

The way that agents communicate (communication protocols) is described in a more detailed level; therefore, it does not affect the design of the architecture. However, some interaction protocols, such as blackboard, influences directly the way that agents interact in the system; as a consequence, it can be an important factor when designing an architecture.

Finally, methodologies analyze a system using different points of view. Methodologies specify interaction protocols; the way agents interact with each other, and how the environment affects the components. Therefore, if a methodology is followed, it can influence the way a system is designed. Depending on the methodology followed, an architecture can be structured in different ways.

4. Agents Standards

4.1. Introduction

The current development in agent systems covers a wide range of technology, paradigms and techniques. The use of the Agent-Oriented paradigm is becoming a common practice for developing applications that require a high degree of autonomy, and the results are very promising. However, one of the main problems that are being faced is the heterogeneity of systems. As a consequence, there are difficulties when systems attempt to communicate and interoperate, and in some cases, reusability is almost impossible.

As a response, different organizations have recognized different needs, and currently they are collaborating in a standardization process. Their main goal is to develop applications using agent technology that can work together. They have successfully achieved the analysis of different aspects of an agent system like interoperability, communication and security.

One of the most important achievements in these standards is the proposal of an Abstract Architecture to support an homogeneous development. Is important to remark, that in their use of the term of Software Architecture is different to the one used in this research. From their point of view, a Software Architecture a set of specifications and guidelines that defines, in an abstract way, elements and their relationships, communication and interoperability between agents [24]

In this work a Software Architecture is defined as a pattern of interconnected components satisfying some structural rules. Moreover, there are many frameworks and toolkits based on these standards. In this work, they are considered as a middleware to support easier development process.

In this chapter, we are going to analyze the FIPA and MASIF standards in order to give a general idea about standardizations and specifications in the field. Additionally, we are going to analyze four frameworks: JADE, ZEUS, Grashopper and COUGAAR; the goal is to give a brief introduction to the development of agent-based systems.

4.2. FIPA Abstract Architecture

The development of an application based on Multi-Agent technology tends to be complex; different organizations develop agents that, because they are heterogeneous, become useless when they interoperate. The foundation for

Intelligent Physical Agents [25], founded in 1996, is an international organization promoting the specification and standardization of agent technologies.

The main objective of FIPA is to specify the interoperation between different agent platforms. Its official mission statement is: "The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings" [47]. It covers almost all the areas related with agent technology, such as architecture, communication, mobility and security [19]. Due to the scope of this work, just the abstract architecture will be analyzed.

The principal goal of abstract architectures is to identify which elements between different technologies are shared and describe them in an abstract way. These elements should be the base for concrete architectures, and because different technologies are sharing the same abstract design, they can interoperate. The proposed abstract architecture (Figure 5) has four elements: *Message Transport, Agent Directory, Service Directory and an Agent Communication Language* [24].

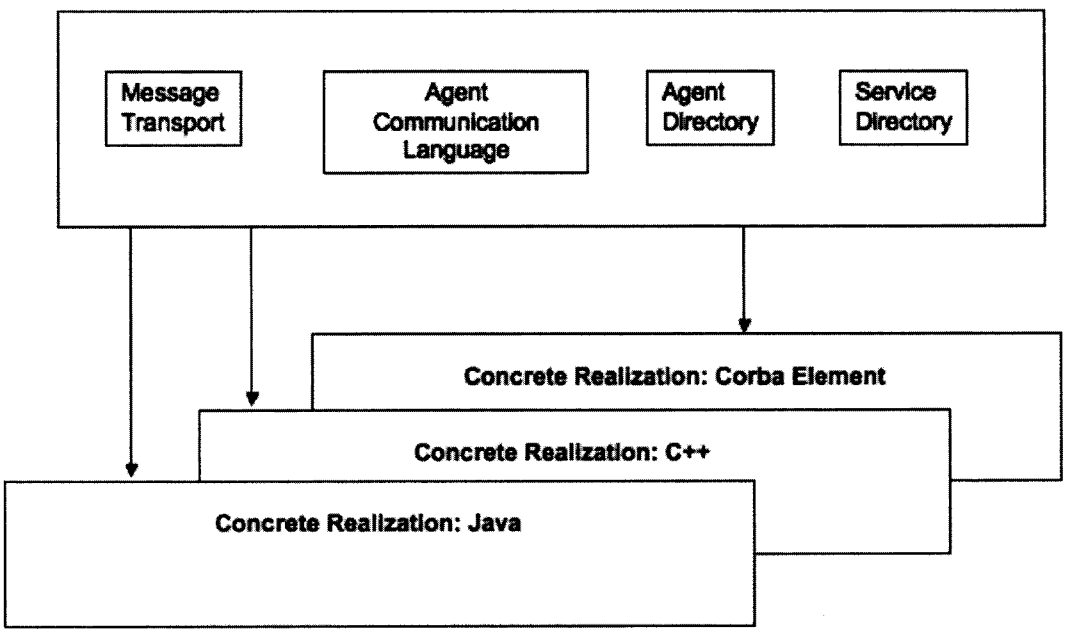

Figure 5. FIPA Abstract Architecture

- **Agent Directory**. When an agent is created, it is registered in the Agent Directory Service. If one agent is trying to reach another one, it can find its location through this service. The agent directory entry is a tuple with

two values: agent name and agent locator; the first one is a global and unique name for the agent; the second one contains the agent's address and description.

- **Service Directory**. An agent registers all of its services in the Service Directory. If an agent wants to use a particular service, it can go to this component and locate the agent that is providing such a service. This component can be thought of as an analogy to the Agent Directory, but instead of being oriented to agent discovery, it is focused on service discovery. The entry is a tuple composed of three attributes: *service name, service type* and *service locator*. The first one is the unique name for the service, the second one sets the category for this service and the last one contains the signature type, service signature and service address.

- **Message Transport**. The communication between agents is possible through messages. A message is composed of three parts:
  - *Message Structure*. It is a key value tuple written in an Agent Communication Language such as FIPA ACL.
  - *Message Representation*. It is represented through a content language such as KIF (Knowledge Interchange Format); the expressions of the content are retrieved through ontologies, which are a common vocabulary of agreed definitions and relationships between those definitions, to describe a particular subject domain. A message contains the name of the sender and receiver that are unique identifiers.
  - *Message Transport*. A message is encoded using the encoding representation necessary for transport and included in a transport message. An envelope is added to the message; it has extra information such as the sender, receiver and additional attributes

- **Agent Communication Language**. The communication services are used in order to exchange agents' messages; they are encoded through an Agent Communication Language (ACL). FIPA proposes FIPA-ACL that is based on KQML (Knowledge Query and Manipulation Language) that describes how to encode a message and its semantics; however, it does not specify anything related with the communication. In addition, FIPA-ACL eliminates ambiguity and confusion, and supports conversations between agents through interaction protocols, which are communication patterns that are followed by every agent.

Finally, it is important to keep in mind that because the architecture is abstract, it cannot be implemented directly; it must be concretized using different platforms like Java, C, or CORBA. There are several platforms that are based in this architecture like Jade, Grasshopper, FIPA-OS, Zeus and Jack. In this work

just Jade and Grasshopper will be analyzed. A more exhaustive comparison of these platforms can be found in [55].

4.3. MASIF Standard

MASIF (Mobile Agents Facility) [35] is a standard based on OMG (Object Management Group) [41] technology that provides a collection of definitions and interfaces for the interoperability of mobile agent systems. Besides the fundamental agency definitions, mobile agents involve additional concepts:

- **Mobile Agent**. Agents that are not limited just to one system; they have the ability to move from one network to another.
- **Agent State**. The execution state of the agent; it is also transported while the agent travels.
- **Agent's Authority**. Identifies where the agent is acting.
- **Agent System Type**. Describes the profile of an agent.
- **Place**. Where the agent is executing.
- **Region**. Set of agent systems with the same authority.
- **Serialization**. Process to store the agent in a way that can be reconstructed.

Even though MASIF attempts to standardize the main points related with mobile agents, there are some parts, such as security and communication, which are in progress or are not addressed in the standard. However, even through the communication specification is outside the scope of MASIF, it is addressed by CORBA. In addition, mobile agent systems are usually written in different languages; the process to convert from one to another is very complex.

One of the goals of MASIF is to format the systems through a similar serialization. In this way, is possible to build bridges between systems in an easier way. Additionally, MASIF mainly standardizes four parts in a system:

- **Agent management**. Specifies standards for the basic agent operations: create an agent, suspend it, resume it and terminate it. In addition, it allows the system to control agents from another system in a remote way.
- **Agent transfer**. Defines the infrastructure to enable agents from different systems to move freely from one to another.
- **Agent and agent system names**. Standardizes the syntax and semantics of the agents' names in the system in order to identify each other. Agent tracking is achieved through MAFFinder, which is the naming service.
- **Agent system type and location syntax**. The agents can locate each other once their location is standardized; therefore, they can communicate and transfer information can happen without any problem.

4.4. Frameworks and Toolkits

4.4.1. JADE

Jade [26] (Java Agent DEvelopment Framework) is a software framework based on the FIPA standardization that acts as middleware for the development of agent systems [6]. It is based on the peer-to-peer communication architecture, where each agent can be viewed as a peer. Among other advantages, Jade provides good interoperability between agents; it is easy to use, and the users can choose the services that want to use [7].

A Jade application is composed of a main container that has different agents and containers. Each container is a Java instantiation that is executed in a single Java Virtual Machine; the agents are distributed when different agents are executed in different hosts. They can communicate with each other using a transportation system based on different technologies such as RMI and CORBA. The Jade architecture is presented in Figure 6.

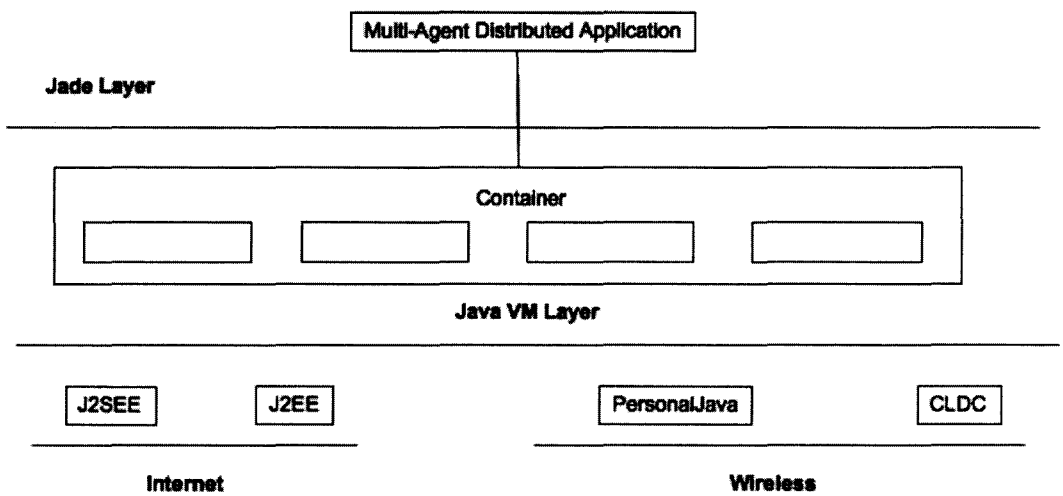

Figure 6. Jade Architecture

Different containers have the same components, and all of them are connected to the main one. The main container is composed of three main parts:

- *Directory Facilitator.* Yellow pages service that supports the registration and retrieval of all the services.
- *Agent Management System.* White pages functionality that manages agent naming services and the agent life cycle management.

- *Agent Communication Language*. Used for communication between all the agents; in this case, RMI is used.

Jade is the most used framework for building Multi-Agent Systems; it has the advantages that it is based in JAVA, it is freeware, and its interface is simple. It is distributed, and it is possible to choose the functionalities that need to be used. Some disadvantages are that Jade does not have enough security mechanisms and that it is still in development; therefore it is not much used in industry. A more exhaustive comparison can be found in [40].

4.4.2. Zeus

Zeus [32] is a tool-kit, based on the FIPA standard, for developing Multi-Agent applications. It emerged as the result of the need of developing methodologies, frameworks and toolkits using agency concepts. Its main goal is to facilitate the engineering of agent applications, to speed up the coding process and to encourage the reuse and standardization of agent technologies.

The environment that Zeus provides helps to configure different agents according to their behaviors. It organizes the agents depending on their relationships, which can be of four types: *sub-ordinate, superior, peer and co-worker*. In addition, the agents are coordinated through protocols like master-salve and contract-net.

Moreover, Zeus follows the FIPA standard; therefore, it provides predefined services such as *name-server agents* (white pages), *facilitator agents* (yellow pages), *a visualizer* and a *communication language*. The first two agents provide the location of the agents and who provides each service respectively. The visualizer analyzes and debugs the system; the communication language that is used is KQML.

In addition, Zeus provides its own type of agent architecture. It is composed of five layers:

- **Definition Layer**. The agent is viewed from an intentional point of view, i.e. through their abilities, goals, resources, beliefs, etc. This layer also defines how the tasks are performed, how the activities are planned and what are the initial facts available to the agent.
- **Organization Layer**. It is viewed in terms of the relationships with other agents. It permits to know who are the other agents in the community and to establish the relation between them.
- **Coordination Layer**. The agent is viewed as a social entity. It defines the strategies used to coordinate the system.
- **Communication Layer**. It manages all the details related with communication issues.

- **API Layer**. Links the agents to the physical components of their resources.

4.4.3. Grasshopper

Grasshopper is a mobile agent development platform, developed by GMD FOKUS and IKV++, used for building distributed multi-agent systems. It is based on the client/server paradigm and mobile agent technology. It is designed following the Object Management Group's Mobile Agent System Interoperability Facility (MASIF) standards and, in its latest version, an extension for the FIPA standardization was added. The Grasshopper architecture (Figure 7) is based on a Distributed Agent Environment, which is composed of three main components: agencies, places and agents. They are grouped by the concept of region, which facilitates the component management [5].

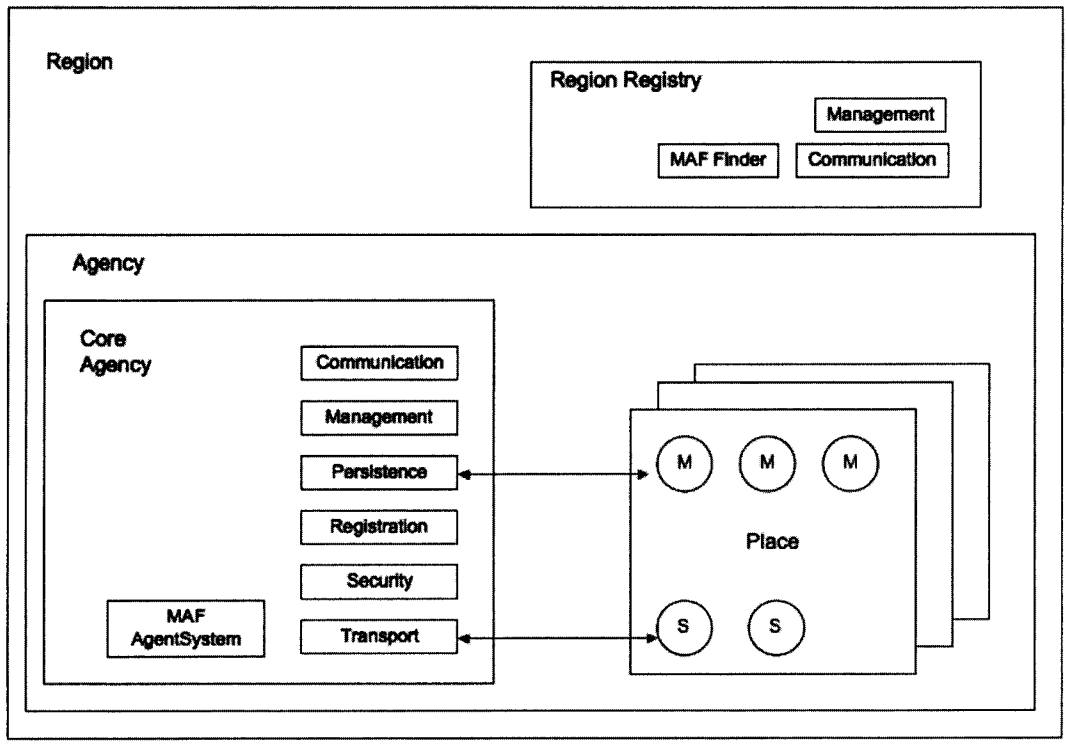

Figure 7. Grasshopper Architecture

- **Agencies**. Associate agents with services that are provided for agent interoperability. The communication between agents is provided by the RMI communication protocol. An agency consists of two parts; the core

agency and one or more places. Core Agencies provides the basic functionality required. The core agency provides the following services:

- *Communication Service.* Responsible for all the interaction between the distributed components. All the interactions are performed via CORBA, Java RMI or plain socket connections.
- *Registration Service.* Lets the agency know where all agents are and where places are hosted. It is connected to the region registry that has information about the agents, their descriptions and the whole region.
- *Management Service.* Allows monitoring and executing agents via operations, such as, start, stopping or removing an agent.
- *Security Service.* There are two security mechanisms in Grasshopper: External and Internal; the first one protects remote interactions between the components; the second one protects resources from unauthorized access.

- **Place**. It groups agents according to their functionality.
- **Agents**. There are two types of agents: mobile agents and stationary agents. Both of them use different services that are grouped in the core agency area; these services provide the minimal functionality required in order to support the execution of the agents.

One of the advantages of Grashopper are that it provides two different standards: OMG and FIPA. In addition, it provides a friendly interface and it is well documented. However, its main disadvantages is that it has weak agent mobility [55].

4.4.4. Cougaar

Cougaar (Cognitive Agent Architecture) [12] is a Multi-Agent Architecture developed by DARPA (Defense Advanced Research Projects Agency). It was originally created as a deployment base for large scale, robust and distributed applications. Cougaar does not follow any particular standard, and it has its own kind of agents which communicate with each other through asynchronous message passing [1].

The main components in Cougaar (Figure 8) are [2]:

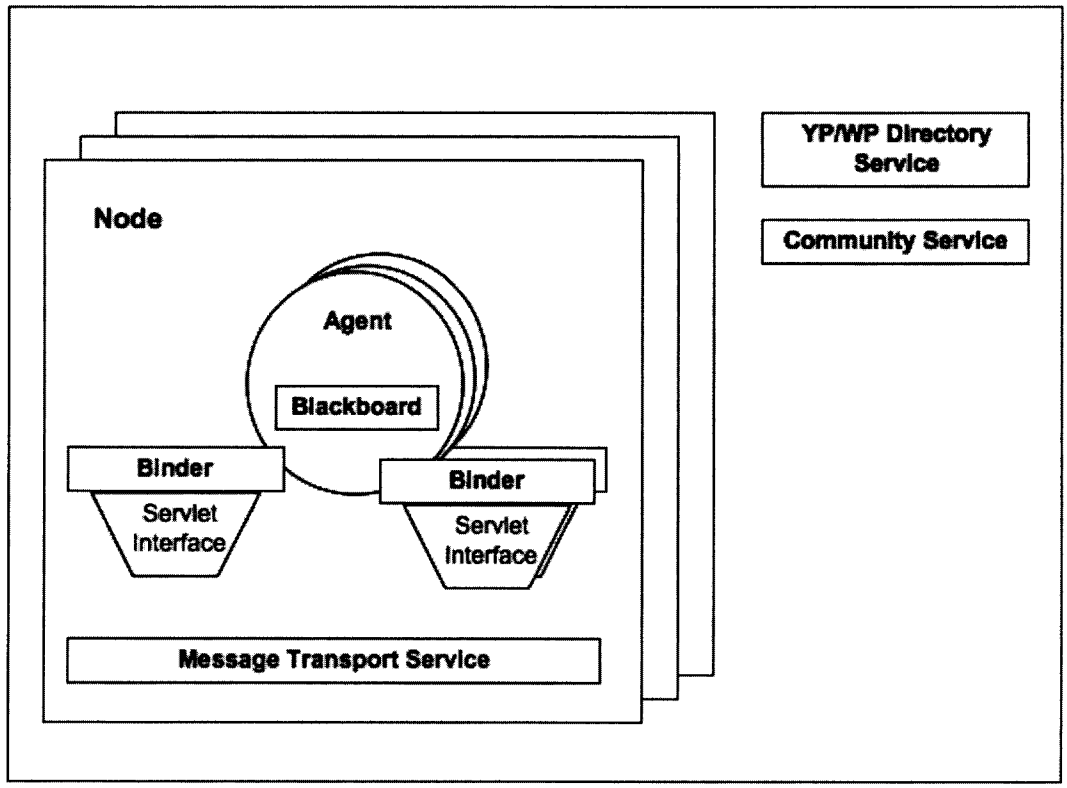

Figure 8. Cougaar Architecture

- **Society**. Set of agents that interact in order to solve a particular or set of problems. It can be composed of different communities and independent agents.
- **Community**. Design concept that groups agents with common functionality. It is composed of different communities and agents. This notion helps in designing the society grouping pieces; however, it is not considered as an architectural concept.
- **Node**. Java Virtual Machine instance that contains multiple agents sharing the same CPU, memory and so on. Additionally, they can be part of different societies and communities.
- **Agent**. Principal unit capable of communicating with other agents. They code and send messages through message transport services or directory services. Messages can also be sent through community services, where the agents and their services can be located. An agent is composed of two parts:
    - *Plug-ins (binders):* Components that add functionalities and services to the agents; they build a wrap around the agent, providing and restricting the services that will be shared with other components.

- *Blackboard.* Establishes the communication between the agent's components. The plug-ins publish and subscribe their objects in the blackboard; therefore, the other components can take the required information from it. The blackboard has three types of objects: tasks, assets and plan elements. Tasks represent a request from one agent to another to perform an operation; assets represent the sources where the tasks are allocated and Plan Elements contain the elements for the tasks.

In the Cougaar architecture, the communities can interact through *a community service component.* If a component wants to be notified of community changes, it should just register with the community service and receive notifications when there are changes. The agents and services can be found through a directory service composed of *yellow pages and white pages services.* The yellow pages service enables the discovery of other agents and the white pages service permits to discover the services. A comparison between Cougaar with similar architectures is presented in [23].

4.5. Relation between Standards, Frameworks and Software Architectures

A standard provides a set of guidelines that help in building a better implementation of a design; however, they do not affect the architecture; they just act as a middleware between the architecture and the design. A framework is based on a standard, and provides tools for an easier implementation; therefore they do not influence in the architectural design too.

For example, MASEL was implemented following the FIPA standard and it was implemented using JADE as a framework. MASEL follows two main architectures: *repository* and *layers* (three tier); however, they just represent the structure of the system. They do not specify the way that agents communicate, where their location is, how it is possible to access to them and how a message can be specified. For that, it is possible to use the standards. Finally, once that all these properties are specified, Jade is used for implement the application in Java.

5. Software Architecture Patterns.

5.1. Architectural Pattern Classification

During the analysis stage of a system, it is possible to get all the functional/non-functional requirements and the general properties needed to design a system. These properties focus a system toward a specific paradigm that adds more characteristics. The design of a system architecture is mainly influenced by these properties together; when a system is based on agents, the main properties that affect the architecture are:

- *Multi-Agent System are composed of Intelligent Agents.* These types of agents present properties that add unique characteristics to the system. The most important are:
  - *Reactivity.* An agent can react when a message is sent directly by an explicit invocation or when it is sent implicitly, trough events.
  - *Proactivity.* Agents decide by themselves the actions that are going to be taken.
  - *Sociability.* Depending on the desired objective, agents interact with the others using different styles. This communication affects the structure of the system, and depending on the possible options, an architecture can vary in many ways.
- *Distributivity.* The location of the agent is one of the properties that influence the design of an architecture. The way that an agent interacts with agents from external systems can be very different to the interactions with the ones that are located in the same system or in local subsystems.
- *Cooperativity.* Depending on which components cooperate to solve specific tasks, an architecture can be modeled in different ways; for example, if every agent provides a partial solution, they can be modeled in one way; but if the functions are not related, they can be structured in a very different way.

Within the literature, there are many systems that can follow different patterns. A system can adopt a pattern according to its properties. Avgeriou proposes classifying the existing patterns according to their architectural view. An architectural view is "a representation of a system from the perspective of a related set of concerns" [3]. A *concern* describes how software components are allocated to the nodes in a network; a *viewpoint* represents a set of elements and their relationships; and *view* is an instance of a system where the elements and relationships correspond to the ones contained in the viewpoint [3].

Based on the Multi-Agent Systems and Intelligent Agents properties, Multi-Agent System architecture can be classified from four main types of view:

- **Layered View**. Decompose complex, large, and heterogeneous systems into interacting parts. The most typical architecture included in this group is Layers.
- **Data-Centered View**. Appropriate when there is common data that is shared by several components. Some architectures included in this group are Active Repository and Blackboard.
- **Component Interaction View**. Focuses on how the components exchange messages and interact among each other. Examples of architectures included in this group are Implicit Invocation and Reactor.
- **Distribution View**. Organizes components' interaction when they are in a networked environment or in different systems. Broker is one the most common architectures included in this group.

5.2. Layers

The Layers architectural pattern helps to structure systems that can be decomposed into groups that have elements with a similar level of abstraction [10]. The layers of the structure are placed on top of each other; the connectors between them determine how the layers interact, and they are defined by protocols.

Layered architectures are mainly composed of *layers*; their responsibilities are to provide services to the next layer (J+1) and delegate subtasks to the previous one (J-1). The layer J always collaborates with the layer J-1; Figure 9 shows a typical layered architecture.

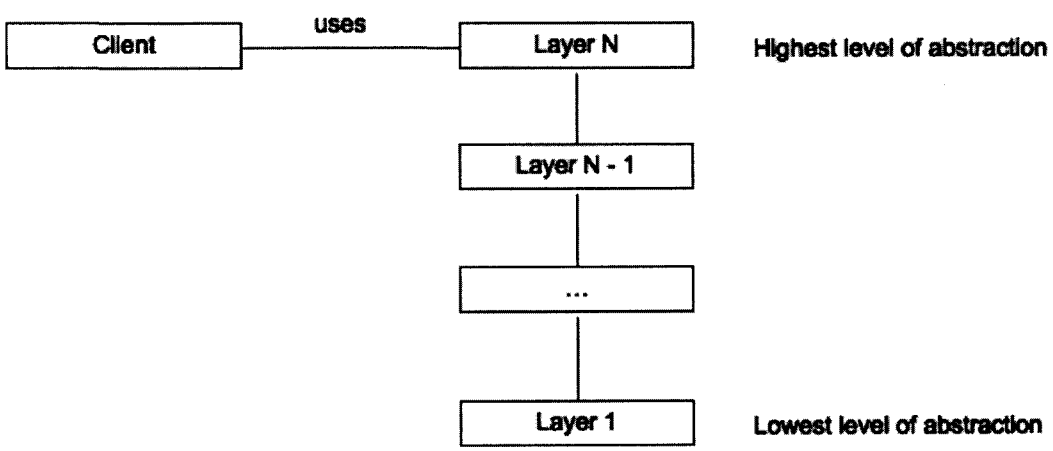

Figure 9. Layers Pattern

The Layer 1, which has the lowest level of abstraction, is considered as the starting layer that is the base for the other layers. The layer J is placed on top of the layer J-1, and finally, the layer N has the top level of functionality. In each layer, all the components must work at the same level of abstraction. Most of the services that J provides, are composed of services provided by J-1, therefore, services depend on services provided by other layers.

The OSI-7 model is a network protocol generally used as a typical layered example. It contains as set of rules that indicates the way that computer programs should communicate across machine boundaries. The OSI-7 model is composed of seven layers [10]:

- *Application.* This is the highest level of abstraction; it provides protocols for different activities.
- *Presentation.* Organizes the information in a structured way and attaches the semantics.
- *Session.* Provides control and synchronization.
- *Transport.* Break the messages into packages.
- *Network.* Selects the optimal route between sender and receiver.
- *Data Link.* Detects and corrects errors in a bit sequence.
- *Physical.* Transmits bits.

Some of the advantages that a layered architecture provides are [3, 10, 53]:

- *Reusability.* If the layers are well defined, using a proper abstraction, the layer can be reused in different contexts.
- *Support for standardization.* If a level of abstraction is well defined and widely accepted; it is possible to develop standardized interfaces. This allows using products from third parties in different layers.
- *Local dependencies.* External changes just affect one layer, and it is possible to modify the affected layers without changing the others. This enhances portability, testability and independence for the components of the system.
- Exchangeability. Individual layers can be replaced by different ones in an easier way. The modifications are minimal, and they are just focused on the connections between layers.

However, the Layers pattern, has some disadvantages [3, 10, 53]:

- *Changes in the behaviors.* If the behavior in a layer changes, it can affect the next ones; therefore lower layers can be shielded from modifications in higher levels.
- *Lower efficiency.* If there are several layers, the data transfer from the top layer to the lowest one can be inefficient.
- *Unnecessary work.* Extra work in different layers can affect the consequent ones and generate duplicated work that is not necessary.

- *Difficulties establishing the correct level of abstraction for each layer.* In some cases, there is not enough clarity in the separation of components; this can generate incorrect levels of abstraction, or force components to be part of an incorrect layer.

A Multi-Agent System (MAS) is composed of elements that, in contrast with other types of systems, exhibit autonomy. However, as a derivation of standard components, agents can be grouped according to the services provided, the way that they interact, etc. A Layers Patterns helps to structure a MAS focusing on the services provided by each agent, the components that use these services and the way that they are sent. However, depending on the system, a Layered pattern may not be enough to design a Multi-Agent System; usually, it complements the use of different patterns in the system.

5.3. Broker

A Broker architecture decomposes the system into a set of distributed and independent elements that can communicate and interoperate between them. These components can be heterogeneous; however, they should be able to interoperate through a communication process that is independent of each component. The communication between components must be in a transparent way; the exchange, addition and deletion of components may occur in real time and the details should be hidden from other components and services [10, 3].

This architecture is applicable when, within an application, there are many clients and many service providers [20]. Moreover, a Broker architecture is based on a layered architectural style [63]. The layers help to structure an application in groups with a very well defined abstraction level.

The main elements that compose a broker architecture are [10]:

- *Client.* Accesses and sends requests to servers and implements user functionality.
- *Server.* Implements objects and services, registers itself with the broker component and sends the responses to the client.
- *Broker.* Communication between the client and the servers, registers and un-registers servers, transfers messages, locates servers and interoperates with other brokers.
- *Client-side Proxy.* Layer that mediates between clients and the broker, encapsulating the system functionality related with the client.
- *Server-side Proxy.* Calls services provided by the server; encapsulates system functionality related with the server and provides a layer that mediates between the broker and the server.

- *Bridge*. An optional component that hides the implementation details when two brokers inter-operate; it encapsulates functionalities of the network and mediates between the system and the remote brokers.

In general terms, a broker is very helpful when distribution is a key role in systems. Among others, the main advantages are [10]:

- *Location Transparency*. Because the components communicate through the broker, clients do not know the servers' locations in order to request a service. In a similar way, servers do not have to know the location of the clients.
- *Changeability and extensibility of components*. Modifications in components do not affect the others; changes in the broker component, for example, do not affect the rest of the system.
- *Portability*. The broker system hides the details using the proxies and bridges; the lowest layers hide the details from the rest of the system; therefore, just these layers must be ported, and not the rest of the system.
- *Interoperability*. Different broker system can interoperate if they have a common message-exchange protocol, which is handled by the bridges.
- *Reusability*. When building a new system, components based on broker architectures can be reused easily; just slight modifications to the proxies are needed.

However, a Broker architecture also presents some disadvantages [10]:

- *Restricted efficiency*. Broker based applications are slower than the ones with a static distribution.
- *Lower fault tolerance*. All the components depend on the broker, if it has any problem, all the system can fail.
- *Testing and debugging*. Because many components are involved, testing can be a hard and tedious process.

A Broker architecture is very suitable in a Multi-Agent System when the structure can be designed from a requester-provider point of view; i.e., an agent requests a service (client) which is provided by different agents (server). When an application is composed of several subsystems, a Broker architecture can help to achieve easier communication and, when concurrency is a main characteristic, the proxy components and the broker can be modeled using the agency definitions.

5.4. Blackboard

The Blackboard pattern is useful for problems that do not have a deterministic solution or known strategy for achieving a solution. In this case, the programs do not interact directly; they are independent and work cooperatively on

a common data structure. The system works with partial solutions that are combined in a solution space; in case that they are incorrect, they are rejected.

This architecture is organized in abstraction levels, which are the conceptual distance from the input and truth of the hypothesis. The lowest one is an internal representation of the input; the highest represents the possible solutions. The system is divided into three components (Figure 10) [10]:

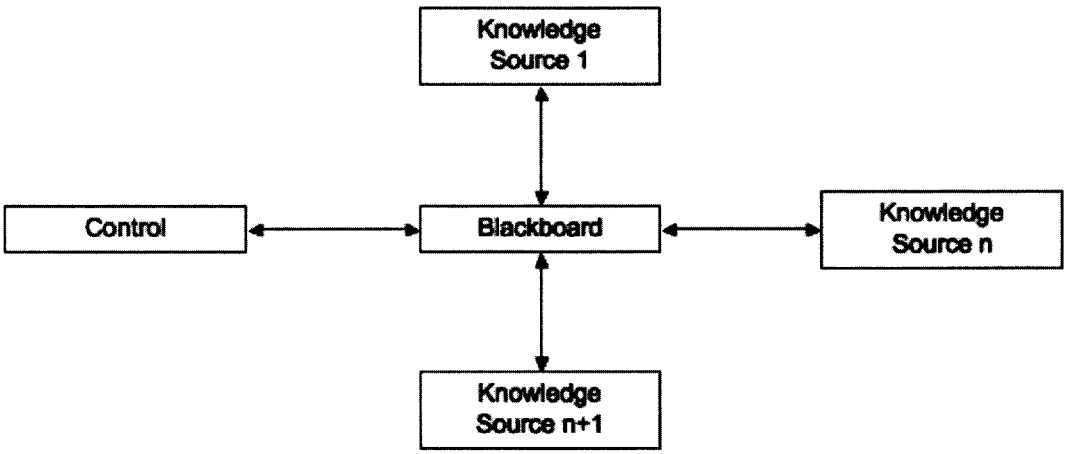

Figure 10. Blackboard Pattern

- *Blackboard.* Where all the data (elements of the solution and control data) is stored. It is composed of a vocabulary, which represents all the elements that can appear in the blackboard, and an hypothesis, which is the possible solution constructed during the solving process.
- *Knowledge Sources.* Independent components that solve specific aspects of the problem. The solution is built through the integration of all the results provided by these components. They communicate through the blackboard; therefore, they have the same vocabulary. They are split into two parts:
  - *Condition part.* Evaluates the current state of the general solution and determines if it is possible to make a contribution.
  - *Action part.* Produces a result that may change the contents of the blackboard.
- *Control Component.* Runs a loop that constantly monitors if there is any change in the blackboard; based on that, it decides the actions that can be taken. Based on the data, it evaluates when a system should be activated.

In addition, there is a knowledge source that does not collaborate directly with the solution, but performs calculations about the control decisions that should be made. The results are named *control data* and are also placed on the blackboard.

Some of the advantages that the blackboard architecture provides are [10]:

- *Experimentation.* When there is not an exact solution, this pattern enables experiments with different algorithms and different control methods.
- *Changeability and Maintainability.* All the components in the system are strictly separated; therefore, if a component is modified, it does not affect the rest of the system.
- *Reusable Knowledge Sources.* Because each component is independent, it can be reused in another system. However, in order to be understood by the others, they must use the same protocol.
- *Fault Tolerance and Robustness.* In this architecture, all the results are hypotheses; just the strongest survive. This provides tolerance for uncertain conclusions.

Some of the disadvantages are [10]:

- *Testing.* Because a system based on this architecture does not follow a particular algorithm, the results are not always reproducible; in addition an incorrect hypothesis can be part of the final solution.
- *Difficulty of establishing the control strategy.* There is not a straightforward way to set the control strategy; it is obtained through experimentation.
- *High development effort.* Due to the trial and error programming when defining the vocabulary and the knowledge sources, the development of systems can take a long time.
- *No support for parallel activation of subsystems.* There is not a defined strategy to activate the execution of knowledge sources at the same time; it depends on the order given in the control loop.

In a Multi-Agent System, a Blackboard architecture is very helpful when the components can be grouped in different subsystems that solve specific tasks. Each subsystem provides a partial solution; all together may provide enough information for reaching the goal of the application. The subsystems can work concurrently and submit the results when they are ready; the control loop will be in charge to activate them when it is necessary and put all the pieces together.

5.5. Implicit Invocation

In a system, components usually interact with the others explicitly invoking sub-routines; however, as a possible alternative, components can communicate with the others in an implicit way through events. An *event* is "a

notable thing that happens inside or outside your business. An event may signify a problem or impending problem, an opportunity, a threshold, or a deviation" [36]. In software applications, when an event happens, it can affect the behavior of the system.

In an event-driven architecture, procedures are not invoked directly; a component announces an event, different components register an interest in this event and associate it with the procedures that will be affected. When an event occurs, all the related elements are invoked in an implicit way.

In this case, a client is just interested in the invocation result, but it can do something else in the meantime [3]. In Multi-Agent Systems, agents are always acting autonomously and when an event occurs, they change their behavior. Agents that react to external stimuli are known as *reactive agents*.

An implicit invocation is composed of modules that signal events without knowing the recipients and procedures that register their interest in specific events. In addition, it is necessary to have an extra element: the event handler; its function is to register, to coordinate, and to activate events. When a process sends an event, it does not know who are the recipients; therefore, there is not a specific processing order; however, it is possible to use explicit invocation as a complementary form of invocation [52].

Among many advantages, implicit architectures [15, 53, 52]:

- Provide a strong level of reusability.
- When a component is added to the system, it is simple to register it for the events in the system.
- It supports for a better system evolution.
- Components can be changed or updated without affecting the interfaces with the other components in the system.

On the other hand, these architectures also provide some disadvantages, such as [15, 53, 52]:

- Components cannot completely control the system.
- When a component announces an event, it is not possible to know if the components will respond to it.
- Exchange of data must be within the events or through repositories; this can affect the performance of the system.
- It is not easy to prove the correctness of an event-based system.

Implicit architectures are very helpful when the system is based on events; since reactive agent systems are very common, therefore, these kinds of architectures are the best option for these cases. This architecture is not very specific about the components that should be included in the structure; they are

just reduced to components and events. However, there are different architectures that are based in these principles, such as the Reactor architecture, that, even though gets into a very detailed level, the main components can be implemented in a straightforward way.

5.6. Reactor

Event based systems can be modeled following different architectures; one possibility is the use of a *Reactor Architecture*, which allows the separation of service request and the sending of them to an application [51]. Even though this architecture is considered as a pattern, several components are described at a detailed level; however, it is possible to apply the main elements to implicit invocation architectures.

In a distributed environment, different clients send many requests at the same time. These requests are identified by indication events, which are the ones that activate different processes in the system. The application should be able to recognize the source of these events and dispatch them to the corresponding service implementation.

As shown in Figure 11, a Reactor Pattern is composed of five participants [51]:

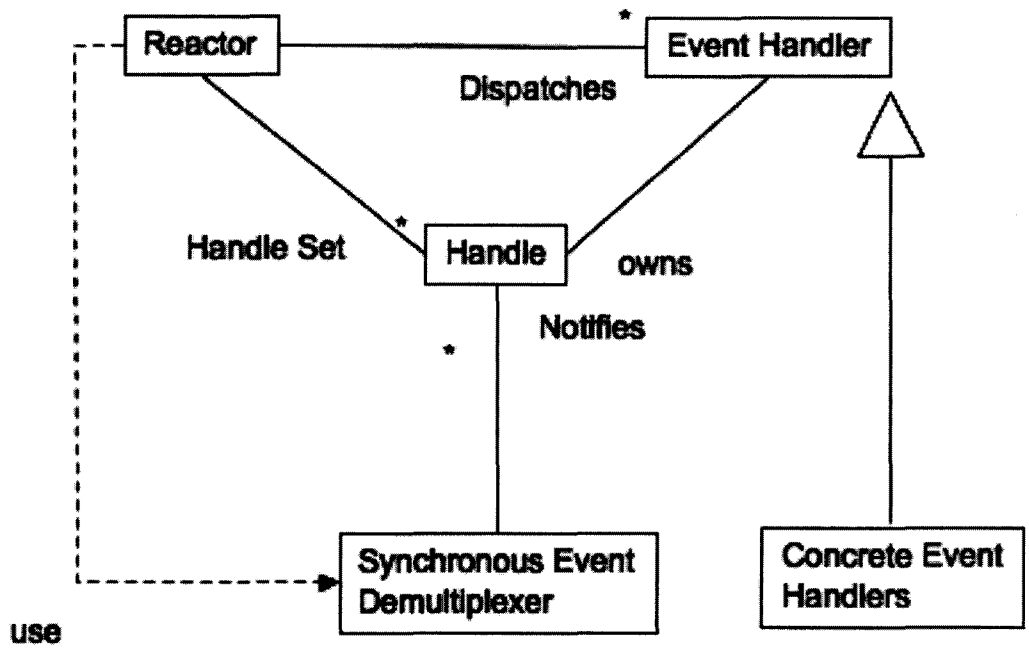

Figure 11. Class Diagram Representation of the Reactor Pattern

- *Handles*. Elements provided by the operating system to identify event sources. Each indication event has an associated handle; when an event occurs, it is queued on its handle and it is marked as ready. Handles can be grouped in *handles sets*.
- *Synchronous event demultiplexer*. Function that is called for waiting for occurrences of indication events on a set of handles. It blocks operations until an event occurs (set as ready in the handle set).
- *Event handler*. Interface that consist of hook methods that represent the operations to process indication events that occur on handles.
- *Concrete event handlers*. Implement the services that an application offers. They implement the hook methods that process the events.
- *Reactor*. Allows applications to register or remove event handlers and their associated handles. It manages handle sets, and runs the application's event loop. When an event occurs, the reactor recognizes who is the requester and dispatches the event to the appropriate hook method.

Among the benefits that the Reactor pattern offers, the most important are [51]:

- Through this architecture, it is possible to separate the events and the modules that will be affected. Therefore, components can be highly decoupled, making them independent and reusable.
- The application can be decoupled into several components, promoting a high modularity. This helps for achieve better reusability, and an easier way to organize the elements.
- Because of the decoupling of the reactor's interface from the operating system, it is possible to reuse this pattern in different platforms.
- It coordinates concurrency control, eliminating complicated synchronization in the application's processes.

However, it also has some disadvantages [51]:

- This architecture does not follow totally the software architecture concepts proposed in this thesis; therefore, depending on the system, its interpretation can lead to some misconceptions.
- Since it is based on events, it is hard to check the correctness of the system, to debug, and to test.

Following the proposed architectural concepts, a Reactor pattern contains two main components: event handlers and a reactor. The way that an event is controlled and represented (handles, demultiplexer and concrete handlers) is implemented in a more concrete level. The way that a method reacts is detailed in

the design of the reactive components, which have the methods that activate the component.

A reactive Multi-Agent system can be viewed as an event-driven application; the Reactor pattern is very helpful for the design of all types of reactive systems. It is more detailed than an implicit architecture; however, it is less flexible. The main advantage provided for Multi-Agent Systems is that this architecture helps to separate the way that the agents send messages or provide services (interpreted as events) and the way that they affect the agents that receive them. In this case an agent provides services without knowing who will use them, and the agents that want to utilize them can register an interest for them, viewing them as events.

6. CASE STUDIES

6.1. Robot Disassembly Process Using a Multi-Agent System

6.1.1. Description:

A traditional manufacturing system usually is designed in a centralized way. Centralized systems present problems when there are failures; they cannot recover from them, or they just simply shut down. These problems are overcome when a system is designed in a distributed way. A good way to model a distributed system is using an intelligent manufacturing environment based on Multi-Agent Systems. This project proposes a design for a disassembly process, which is a special case in manufacturing systems [43].

The disassembly system is divided in three parts:

- *CAD Model.* Analyzes the parts of the model, processes data and sends the results as input for the Supervising Cell.
- *Supervising Cell (3IDS).* Analyzes the information from the input and sends a possible disassembly plan to the Disassembly Cell.
- *Disassembly Cell.* Contains robots that execute the operations; in this case this part is simulated.

This project just analyzes the 3IDS part, where the Multi-Agent System is designed to work on a predicted scenario. The system is composed of two main areas:

- *Sequence Analysis.* Searches for the best disassembling sequence.
- *Disassembly Process.* Simulates the proposed disassembly process.

6.1.2. Initial Design

The architecture is divided in three logical units: data structures, decision-making unit and control unit.

- **Data Structures**. It does not contain agents, but, has all the set of structures that work as inputs for the other logical units. It has the information obtained from the CAD model, such as the set of possible sequences for disassembly.

- **Decision Making Unit**. Using the data structures, it chooses the set of best sequences to perform the necessary operations. If an operation fails, it chooses another one or calls for the intervention of the supervisor. This unit is composed of three different agents:
  - ***Decision Agent***. Based on the information acquired from the operation agents, it decides which one is the best one. It is based

on states; the initial state is the dismantling sequence. In each step, the agent tries different operations; depending on the obtained results, it changes its state. On a collision, the decision agent analyzes different alternatives and creates different agents for the possible operations that can be executed.

- ***Operation Agent.*** Agent created for each possible operation. It contains the method *negotiate*, which returns the first potential operation that will be executed by the agent.
- ***Action Agent.*** Agent created after a decision for performing the chosen operation is taken. It travels between the *decision-making* and the *controllers unit*; if the controller has a problem when executing an operation, this agent goes to the decision unit where another decision is made or cancels the agent. These agents are placed in an agent container.

- **Control Unit.** Unit composed of a set of agents that execute the operations chosen in the decision unit. Two objects are contained in this unit: the *robot* and the *operator*, which are controlled and executed by the controller agents. Each agent searches for an action agent in the agent container; if the action is executed, then the action agent is disposed of and the decision unit creates the next action agent. Three types of agents are included in this unit: *Station Controller Agent, Supervisor Agent* and *Operator Agent*. They assist persons who interact with the system.

6.1.3. Architecture Analysis

The designers do not clearly state the architecture that is used. They describe the components and their relationships, but do not present any graphical representation of the initial design being followed. However, through the provided description, it is possible to get this representation.

Moreover, as shown in Figure 12, it is possible to design the system following a classical multi-layered architecture. In this case, each component is represented as a box and each connector as an arrow. The robot and operator, even though they are not agents, are considered as part of the architecture. The layers that comprise the architecture are:

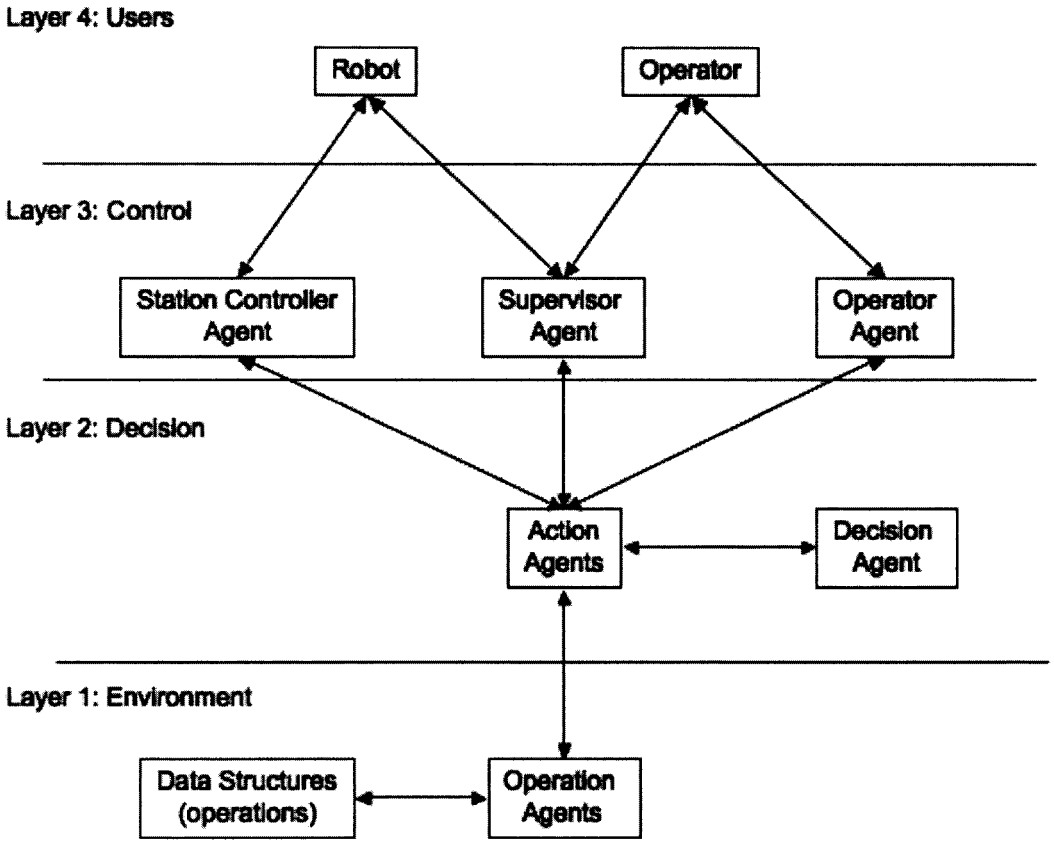

Figure 12. Robot Disassembly Process

- **Layer 1: Environment.** Includes the information received from the CAD model and the operation agents, which represent all the possible operations from the data structures.
- **Layer 2: Decision**. This layer includes the decision and action agents; in this level, the strategy to follow is planned. Decision agents are a subset of the most suitable operations, and the action agents perform the chosen operations.
- **Layer 3: Control.** Composed of the controller agent, supervisor agent and operator agent; they communicate with the actions agents that are in the repository and execute the operations. These agents provide services and suggest a disassembly plan that the users, who are in the next layer, can use.
- **Layer 4: Users.** This layer is composed of robots and operators that, even though they are no humans, are considered as users because they request and receive services from the next layers.

The initial design of this system was not structured using the proposed architectural concepts. However, through the analysis of its different components, it was possible to model the system as a multi-layered architecture. A multilayered architecture helps to decompose the system according to the functionality of its components. Each group can be decoupled from the others and can be organized into groups that can provide services to the others in a hierarchical way.

6.2. MASACAD: Multi-Agent System for Academic Advising

6.2.1. Description

MASACAD (Multi-Agent System for Academic Advising) [18] is a Multi-Agent System that advises students using a machine-learning paradigm. Internally, it uses different techniques for user modeling, such as e-learning, information customization, agent systems, machine learning and web mining.

Currently, just the course registration advisor has been implemented; depending on the students' profiles and external factors, such as room capacity, pre-requisites, extra courses, etc., the system advises courses that fit the students' requirements for the term. In the current version, students give the information to the system; therefore it does not attempt to learn from their desires. This version just focuses on learning how to perform the advising process.

The application is mainly composed of three components:

- **User System.** It is in charge of the interaction with the students; it presents the graphical interface, receives the queries from the students and retrieves the required information.
- **Course Announcement System**. Interacts with the course server through the Internet in order to notify any changes and requests related with the courses.
- **Grading System**. Receives queries related to the student and retrieves the requested information.

6.2.2. Initial Design

This case follows the architecture proposed in the Bee-gent (Bonding and Encapsulation Enhancement Agent) framework [29]. It is focused on distributed systems providing coordination and obtaining consistency in the behavior between different systems, software packages and so on. Bee-gent is composed of two types of agents that are coordinated through Internet Protocol messaging:

- **Agent wrappers**. Provide an agent interface for existing applications, wrapping sub-systems and connecting to the network.

- **Mediation agents**. Implement the interaction between agents, as well as mediating, coordinating and managing the communication and interoperability between applications.

In this application, each system has its own agent wrapper that coordinates the communication with the other systems; agents as represented as black circles and the external systems as boxes. Figure 13 shows the original architecture.

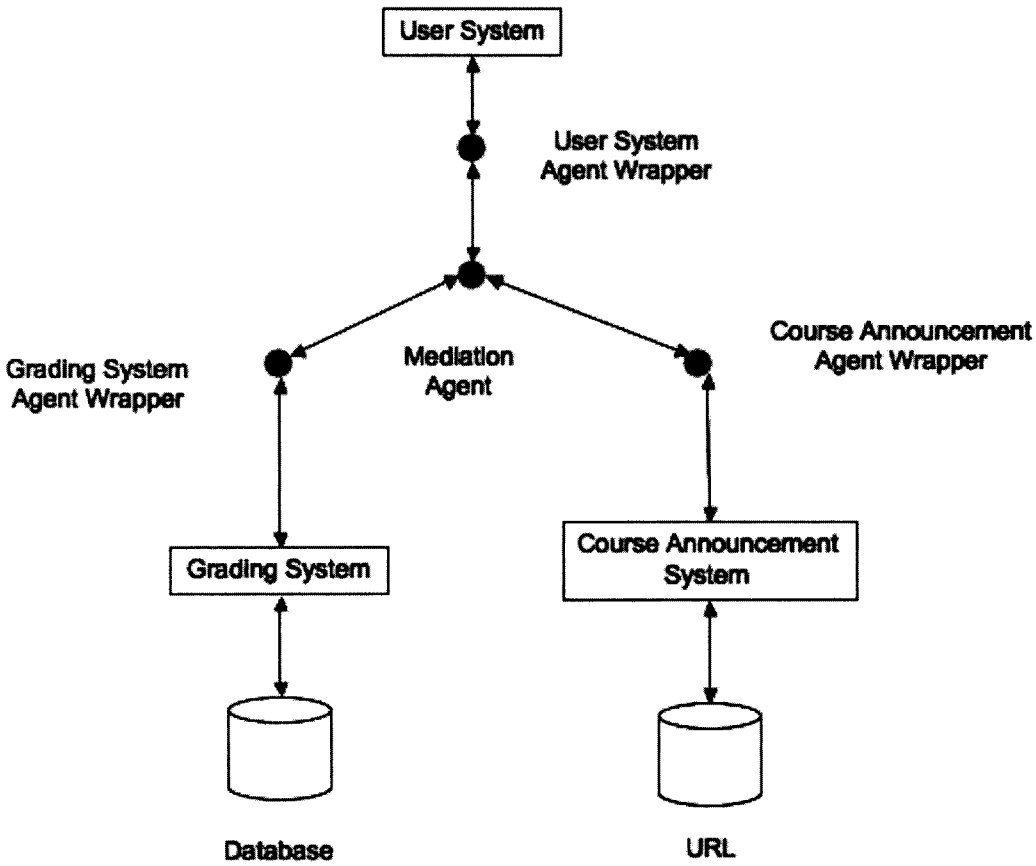

Figure 13. MASACAD Architecture

All the agent wrappers are coordinated through the mediation agent. The following agents comprise the system:

- **Mediation Agent**. It travels between the three systems in order to request and retrieve information. The systems do not send the information directly to the others; everything is coordinated through this agent.
- **User System Agent Wrapper**. It communicates with the user through a graphical interface, and allows him to express what he wants or to modify

something. Finally, it returns the results to the user. In addition, it creates a mediation agent in order to interact with the other agents.

- **Grading System Agent Wrapper**. It is in charge of the information management, information access and information retrieval related to the students. It receives a request from the mediation agent, and gives back the results.
- **Course Announcement Agent Wrapper**. It interacts with the web-server. It receives a request from the mediation agent and then establishes communication with the system at the address specified by the URL, extracts the information and finally sends back the results.

6.2.3. Architecture Analysis

Even though MASACAD follows an architecture based on a proposed framework, it can be modeled and enhanced using well known patterns. The system architecture can be modeled from three different points of view: user interaction view, which analyzes the way that a user interacts with the system; layered view that organizes components in a hierarchical way according to the services provided; and distribution view because there are subsystems. However, the interaction view involves the User System component that is considered as an external system; therefore, that view will not be considered in this analysis.

From a distributed view, the component does not have direct communication; they are intercommunicated through the wrapper and mediation agents. In addition, the subsystems are highly decoupled and located in different networks; therefore, in this case, the *broker pattern* is well suited. This architecture is used to structure distributed components that interact through remote interactions. These components are independent and coordinated by the broker component.

In addition, a broker architecture is based on layers. The Layers Architectural Pattern helps to structure an application, decomposing it in sub-groups that execute similar tasks having a particular level of abstraction. The functionality of this structure depends on the way that the layers communicate with each other. In this case clients request services to a server; however this communication is not direct. A broker can be viewed as a layered client-server, where the communication is not direct.

In this case, the communication is clearly separated from the application functionality. All communication in the system is hidden through the mediation agent (broker) that has a client side to construct the invocations and a server side that invokes the operations. The communication between the broker and the systems is done through the wrapper agents, which in the terminology of this architecture, are named proxies. The proposed architecture is shown in (Figure 14).

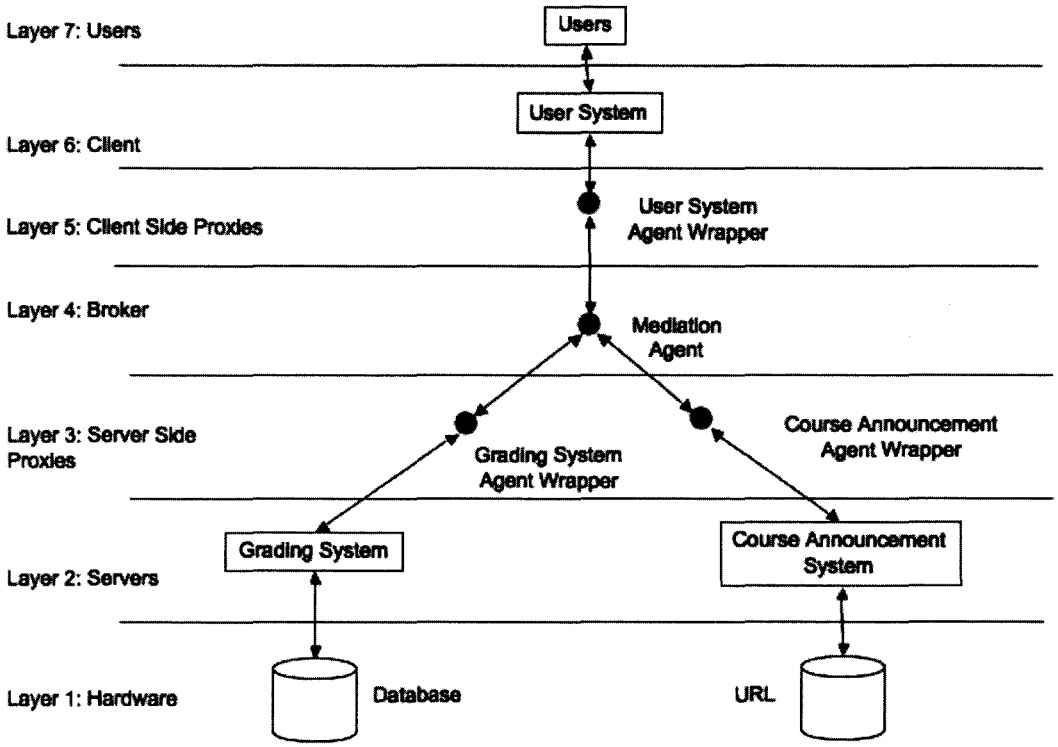

Figure 14. MASACAD Architecture (Broker and Multilayered)

In this case, the components of the broker architecture, also represented as a layered architecture, are:

- **Layer 1: Hardware Devices**. It presents the lowest level of abstraction; in this case it is comprised of the database corresponding to the grading system and the URL database. The next layer requests information, and it just sends back the results. This component is not included in a broker architecture; however, it can be part of a system.

- **Layer 2: Servers**. They implement specific services, receive the requests from the clients, get the information from the corresponding databases and send responses through the Grading and Course Wrappers. In this case this layer is comprised of the Grading System and the Course Announcement System.

- **Layer 3: Server Side Proxies**. They request information from the Grading and Course Systems and encapsulate their functionalities. They act as mediators between the Grading and Course Systems and the Mediation Agent.

- **Layer 4: Broker**. It is the coordination layer; the main function is to coordinate the systems establishing a communication between the client (User System) and the Servers (Grading and Course Announcement System). The broker component in this case is the mediator agent.

- **Layer 5: Client Side Proxies.** They encapsulate User System functionalities and mediate between the User System and the Grading and Course System. They mediate between the broker and the client components.

- **Layer 6: Client.** The component that mediates between the users and the system; it sends requests through the User Wrapper. In this case, it is comprised of the User System.

- **Layer 7: Users**. The highest level of abstraction in the architecture; they are the students who use the system.

This case is based on the Bee-gent framework, which provides a broker-based architecture; therefore, MASACAD was designed following the proposed architectural concepts. The main difference between the original and broker architecture is that Bee-gent proposed an agent wrapper for each subsystem that is a variation of the client/server side proxies in a broker architecture.

MASACAD was very well suited for the broker architecture, which is very helpful when different systems are added. It also decouples components and provides a high degree of reusability. A broker-based system can also be decomposed in different layers; this provides a clear communication between components that are organized in a hierarchical level.

6.3. *MASEL: Multi-Agent System for E-Learning and Skill Management.*

6.3.1. Description

MASEL [16] is an e-learning system used in the industry. It creates personalized training paths for each employee, measures each worker's information, and tries to reduce learning gaps between users. The system assigns individual objectives, controls the knowledge acquisition in an adaptive way and manages a *skill map* for the organization. A skill map stores employee data such as information about their roles in the organization, their current knowledge level, and the knowledge level relative to other workers.

Educational contents are represented as *learning objects*, which are independent units that can be combined for creating personalized learning paths.

Some examples of learning objects are documents, slides, pre-recorded lessons, and so on. They can be related through *Learning Object Metadata,* which classifies them with respect to their objective, topic, media used, etc. This data is modeled using XML, a meta-language that facilitates the sharing of data from different systems that are usually Internet based.

Within the skill-managing context, it is important to individualize learning objectives and evaluate them in order to determine knowledge gaps. These gaps are filled by proposing different courseware based on the information acquired from the database that is composed of the learning objects. The courseware is adapted according to student improvement, and consequentially, a bridge between individual and organizational goals is created. All the student's improvements are managed and updated through the skill map.

The main goals proposed in the MASEL e-learning system are to:

- Help *Chief Learning Officers* define the learning strategy based on the roles and competencies required for the organization.
- Manage the organization's skill map.
- Measure competency gaps.
- Support employees in filling their learning gaps.
- Enrich courseware through personalized learning paths.
- Assist Chief Learning Officers in choosing the best employee for a given role.

6.3.2. Initial Design

MASEL is implemented using JADE, which follows the FIPA standard, and the information is modeled using XML. It does not follow any particular architectural pattern, and the interoperability between agents is direct. The architecture is shown in Figure 15. In this case all agents are represented as boxes, their connectors and arrows and the information sources as databases.

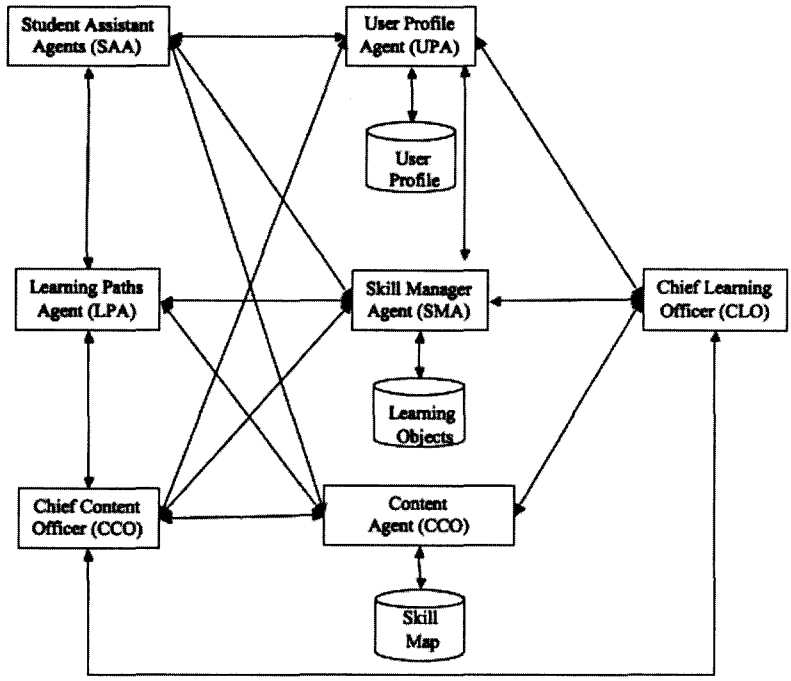

Figure 15. MASEL Architecture

The system is composed of seven agents:

- **Chief Learning Officer Assistant Agent (CLO).** Helps the Chief Learning Officer to define roles, to associate competences, and to specify the level of knowledge required for each role. It requests of the Skill Manager Agent (SMA) the most suitable employee for a role; solicits the learning activities history to the Content Officer Assistant (COA); and presents the received information from the Chief Learning Officer. Its main function is to be an interface between the officer and the system.

- **Student Assistant Agent (SSA).** Assists each student in order to fill his/her gaps. It requests a learning path to the *Learning Path Agent (LPA)* and presents it to the student; sends the assessment feedback to the LPA; with the help of the Skill *Management Agent* (SMA), shows information about competence gaps; and with the help of the *User Profile Agent* (UPA), it manages the learning activities history.

- **Learning Paths Agent (LPA).** Creates learning paths for each student; provides pre-assessment tests for evaluating gaps; selects learning objects for building the path; and modifies the courseware based on the feedback received from the student. It gets the information related with the skill map from the SSA and the one related with the learning objects from the *Content Agent (COA).*

- **Chief Content Officer Assistant Agent (CCO).** Supports the content agent in order to manage the database. It shows the learning history in collaboration with the User Profiles Agents and executes operations in the Learning Objects Database with the help of the Content Agent.

- **Skill Manager Agent (SMA).** Stores and retrieves information, executes operations, and manages all the things related with the Skills Map Database.

- **Content Agent (COA).** Manages the Learning Objects Database; inserts, updates and deletes learning objects; and executes queries related to Learning Objects.

- **User Profile Agent (UPA).** Stores the information about users, such as personal data, learning activities and log-in information; updates competence levels according to the learning activities with the help of the Student Assistant Agents and the Skill Manager Agent; and acts as the log-in interface for the users.

6.3.3. Architecture Analysis

From a data point of view, the system is divided in two parts: database and components. A *database* is a centralized structure that stores information related with the system that can be used to make decisions. A database system is a typical example of a *Repository Pattern*, which is a variation of the Blackboard Pattern. In contrast with the blackboard style, the central architecture is not a control system; it is the repository, which is controlled by users or external systems [10].

In this case, the repository architecture is composed of the following elements:

- Repository. In this case the database is distributed in three parts: user profile, learning objects and skill map; however, they can be viewed as a single unit. In this case, the database is comprised of the three databases from the initial design: User Profile, Learning Objects and Skill Maps Each database has its own manager, which is also considered as part of the repository. Therefore, in this case, the User Profile Agent, Skill Manager Agent and Content agent are also part of the repository
- User System. This system is a client that gets all the information relevant to the employees; it includes the Student Assistant Agents and the Learning Path Agent.
- Chief System. This system interacts with all the information related with the chief officer; it is composed of the Chief Content Officer and the Chief Learning Agents

In the initial design, the Chief System communicates with the user system for accessing the database. This communication is redundant, because each system has direct interaction with the database; therefore the intercommunication between systems can be eliminated. In Figure 16, this possible communication is presented as a dashed arrow.

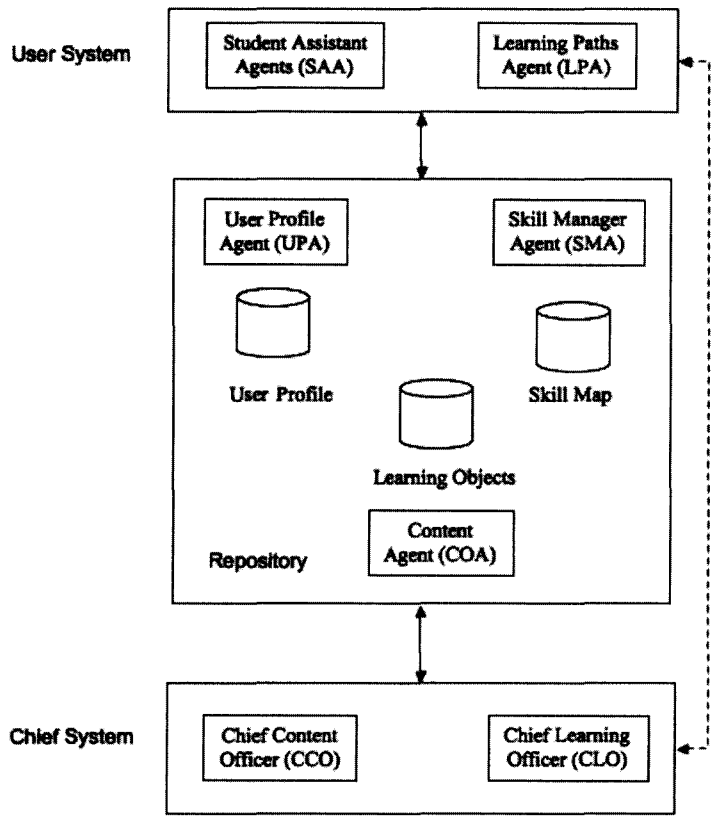

Figure 16. MASEL Architecture (Repository)

Moreover, from a multilayered architecture point of view, the system can be structured as a three-tier client-server architecture (Figure 17). In a client-server architecture, two independent components need to communicate: client and server. One or more clients ask for services that the server provides; a three-tier architecture is a special case of multi-layered architectures that focus on a client-server architecture. The layers that compose a three-tier architecture are:

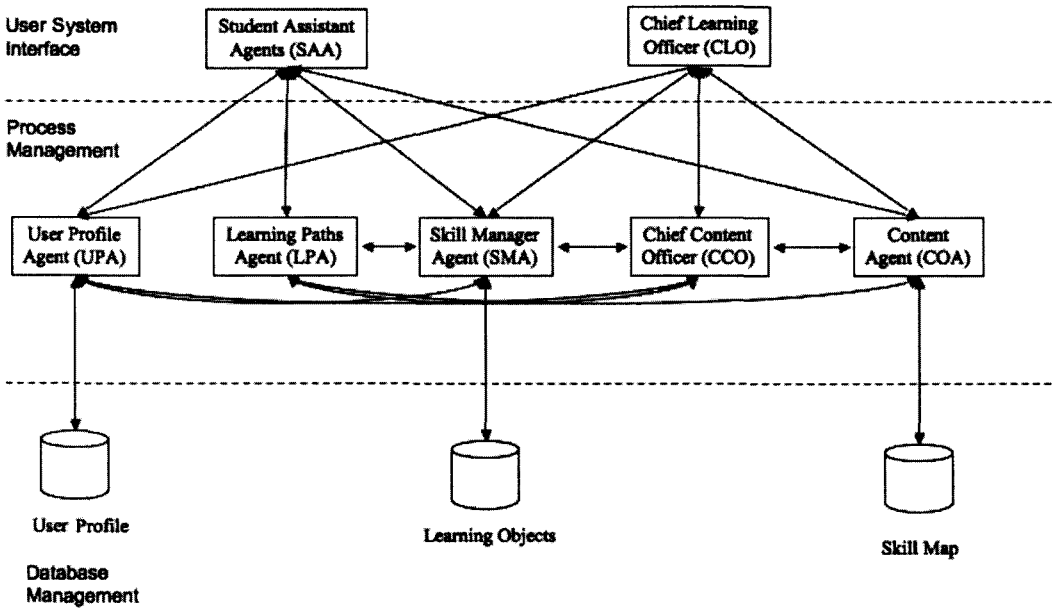

Figure 17. MASEL Architecture (Client – Server)

- **User System Interface.** Layer that acts as an interface with the users. The agents that provide services to the users are: Agent Assistant Agents and Chief Learning Officer. These agents do not interact directly with the database; they establish communication with the next layer, which queries information from the databases.
- **Process Management.** Executes all the operations related with the users and interacts with the databases retrieving the necessary information. It is composed of the User Profile Agent, Learning Paths Agent, Skill Manager Agent, Chief Content Officer and Content Agent. All of them provide services to the User System Interface and get the information from the Database Management Layer. All of these agents interact with the next layer and provide all the services that the users interface need.
- **Database Management**. Where all the information is placed; it is comprised of the User Profile, Learning Objects and Skill map and has the lowest level of abstraction.

The original design of MASEL does not follow any specific architecture; however, through this analysis, two architectures can be implemented. The architecture was obtained from a data point of view and the client-server architecture from a multilayered point of view. In addition, the multilayered architecture complements the repository providing an easier understanding and a reusable architectural design.

Through this case, it is possible to conclude that, even though originally the application was not based on the proposed architectural concepts, using them simplify things considerably. Through these patterns, it was possible to discover some redundant communication between some agents; the communication with the database was defined in a clearer way, therefore the system has a stronger structure that can be used in cases where there is an interaction between clients and servers that share information through a common database.

6.4. Telemedicine for Diabetes

6.4.1. Description

E-medicine is a field of Information Technology applied to medicine that has become very popular in the past few years. It integrates information, communication, and human-interface technologies with those related with health and medicine. E-medicine is usually used in four areas: lifetime health and medicine, personalized health information, teleconsultation and continuing medical education.

Among other services, e-medicine remote distant medical services and clinical practice. Additionally, it establishes medical databases, exchanges medical information, and provides security, privacy, efficiency, convenience and reusability. A very common practice in the e-medicine field is the use of diagnosis tools that are used to help physicians to analyze patients. They guide them with the therapies that should be followed and the medicines that should be taken; they are connected with different resources and they help physicians to acquire new knowledge. They also provide information for the patient explaining the treatment that should be followed.

This system presents a special case study "Telemedicine for Diabetes" that shows an e-medicine system using the multi-agent paradigm. Telemedicine uses communication technology in the provision of healthcare; in this case it helps to manage the healthcare process for diabetic patients; provides real time monitoring and contacts immediate resources for therapy. The system is implemented by the diabetes department in hospitals, and it provides immediate medical services, therapy and consultation. Additionally, it integrates other systems for education, training, management, security and databases [56].

6.4.2. Initial Design

The proposed architecture (Figure 18) does not follow any particular pattern; however, it groups all the components into sub-categories according to their functionality. In this way, the architecture is similar to a relaxed-multilayered architecture. The components are grouped in four categories: Control, Implementation, Interface and Environment groups; the communication

between the first three groups is internal and for the last one, it is external. In this case all components are represented as boxes and their connectors as arrows; however, it is important to notice that not all boxes are agents. In case they are agents, the name of the component has the word "agent" explicitly.

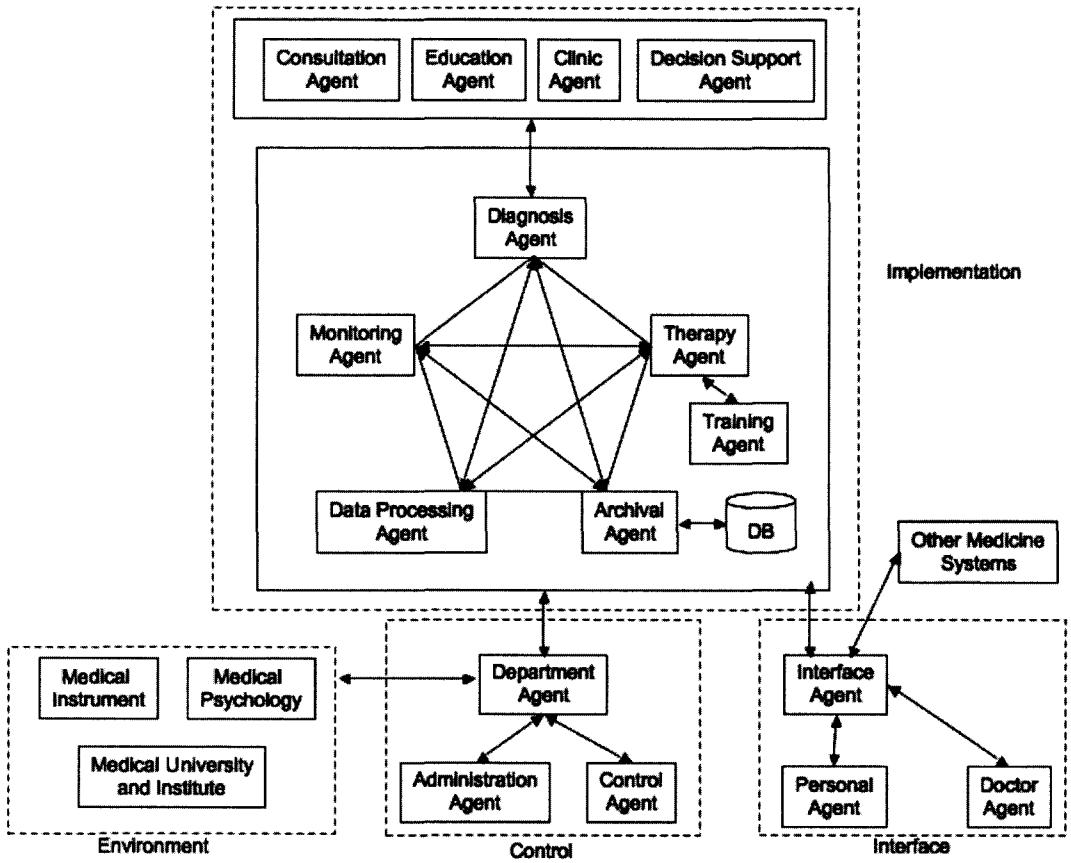

Figure 18. Telemedicine for Diabetes (Original Architecture)

6.4.2.1. Environment Group

This group is composed of the external departments, such as medical instruments, medical psychology, medical University & Institute, and telecommunications. They interact with the system through the department agent that is part of the control group. These departments are not considered as systems, they are just external resources. The interface agent is located in the interface group, it coordinates the external systems which are considered as an external group.

6.4.2.2. Control Group

The agents that are part of this group are the ones that control the system, handle conflicts between agent communications, manage decisions and assign work to other agents. They coordinate the implementation group and also interact with the environment. The agents that comprise this group are:

- **Department Agent.** Controls the local information acquired from the environment for specific departments and sends it to the administration agent.
- **Administration Agent.** Using the information received from the department agent, it assigns tasks to the agents.
- **Controller Agent.** Controls the system, coordinates the agents and mediates conflicts among them.

6.4.2.3. Interface Group

This group includes all the components that link the users and other systems. Users are not defined in the system; however, they are implicit as part of the environment. Agents included in this group are:

- **Doctor Agent**. Manages schedules and appointments.
- **Interface Agent.** Mediator between the e-medicine system and other sources. It links the patient with other systems, provides search and information systems and provides interaction with the doctor, personal and security agents.
- **Personal Agent.** GUI between the users and the system.

6.4.2.4. Implementation Group

This is the core part of the system; agents included in this group have particular tasks and execute internal operations that will help in order to reach a common goal. Agents included in this group are:

- **Monitoring Agent.** Monitors agents and transmits information to the data processing agent.
- **Data Processing Agent.** Integrates and processes the monitored data.
- **Diagnosis Agent.** Analyzes the situation with the information available and makes a decision/suggestion for the patient.
- **Consultation Agent.** Consults the enquiry of patients and contacts with the diagnosis agents.
- **Education Agent.** Provides e-learning and introduces new information for physicians.
- **Decision Support Agent.** Integrates knowledge and provides decision approaches to the diagnosis agent.
- **Therapy Agent.** Defines a proper therapy.
- **Training Agent.** Explains to patients how to follow the therapies and how to take medicines.

- **Record (Archival) Agent.** Archives patient records and updates patients' databases.

6.4.3. Architecture Analysis

Even though the original architecture was designed without a particular style, it can be viewed as a relaxed variation of a multi-layered architecture. A *relaxed layered system* is less restrictive about the relation between layers, therefore each layer can use services of all layers below it [10]. In this case, each group in the system can be represented as a layer. However it is not clear their abstraction level.

Following a layered architecture (Figure 19), elements in this system can be grouped in a hierarchical way according to services provided. In this case, the components that comprise the layers are:

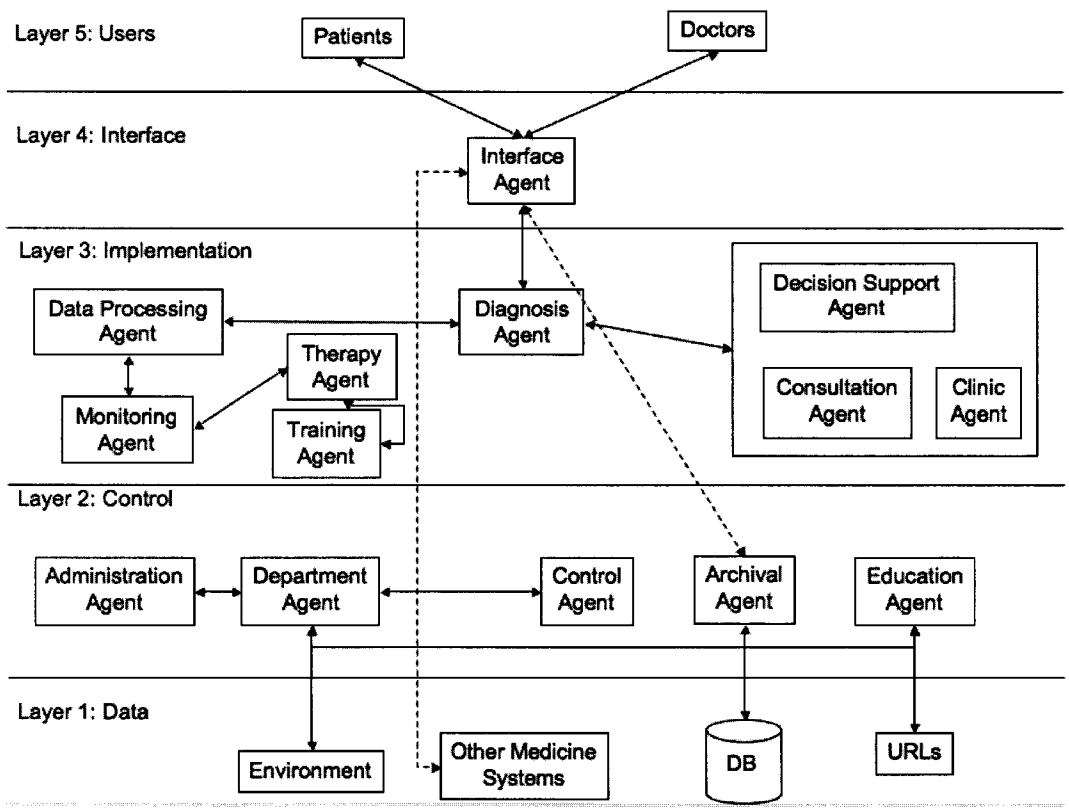

Figure 19. Telemedicine for Diabetes (Multi-Layered Architecture)

- **Layer 1: Data.** The first layer, which has the less abstract level, contains all information sources: environment, other medicine systems, the database and the URL's. They provide the information services to the control layer.
- **Layer 2: Control.** This layer uses services provided by the information layer and sends the information to the next layer. It is composed of the Administration Agent, Department Agent, which gets the information from the Environment, Control Agent, Archival Agent, which interacts with the database, and the Education Agent, which communicates with e-learning resources.
- **Layer 3: Implementation.** It comprises the main operations of the system. Agents included in this layer receive requests from the users, execute some operations and returns a solution. The agents that compose this layer are the Data Processing Agent, Monitoring Agent, Therapy Agent, Training Agent, Diagnosis Agent, Decision Support Agent, Consultation Agent and Clinic Agent. All of them provide services to the users, who interact using the interface services provided in the next layer.
- **Layer 4: Interface.** This layer is composed of the interface agent that communicates the users with the systems.
- **Layer 5: Users.** This layer presents the most level of abstraction and it is composed of the users, which are Patients and Doctors.

It is important to remark that the interface agent uses services from layers that are placed different levels below. In a strict way, this is not possible. However, there is a variation of this architecture, called flexible layered architecture, which permits this interaction. Another possibility is to use new components and placed them in the next layers in order to realize all the intermediate operations.

The system has two types of users: patients and the doctor. A patient expects to be diagnosticated presenting a set of symptoms or characteristics, and to receive a therapy as a result of this consultation. In addition, the patient expects to receive training about how to follow the therapy. On the other hand, a doctor expects to monitor and consult the information related with the patient and to communicate with external systems and resources.

From a data-centered view, it is possible to use a *Blackboard Architecture* [3, 10, 53]. This architecture has three components: *knowledge sources, control system and blackboard.* The system can be decomposed into many subsystems that provide partial solutions; however, they differ from the groups proposed in the original architecture. Moreover, it is possible to use the control component proposed and add an extra component, the *blackboard*, which communicates with all the sub-systems (Figure 20).

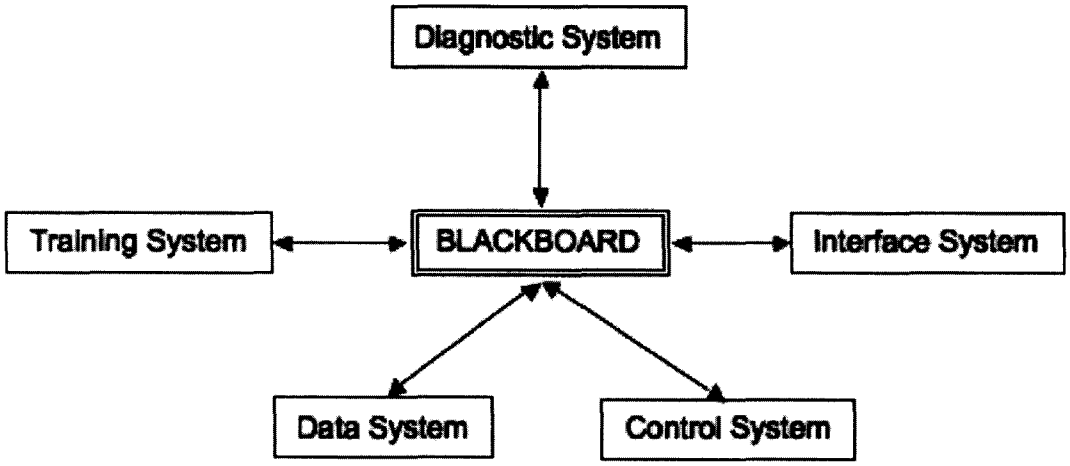

Figure 20. Telemedicine for Diabetes (Blackboard Architecture)

The system is decomposed into the following knowledge sources:

- **Interface**. Interacts with users and external systems through the interface agent. Patients communicate with the system through the personal agents, which provide the graphical interface. In contrast with the original architecture, this sub-system also includes the Education Agent, which communicates with external references. Additionally, it includes the security agent and doctor agent, which execute specific tasks that are not related with the rest of the system. Figure 21 shows the structure of the interface knowledge source. The blackboard component is represented as a box with double-lined border. Agents are represented as a non-colored single lined box; external information sources are represented in grey, and all the subsystem is grouped in a dashed box.

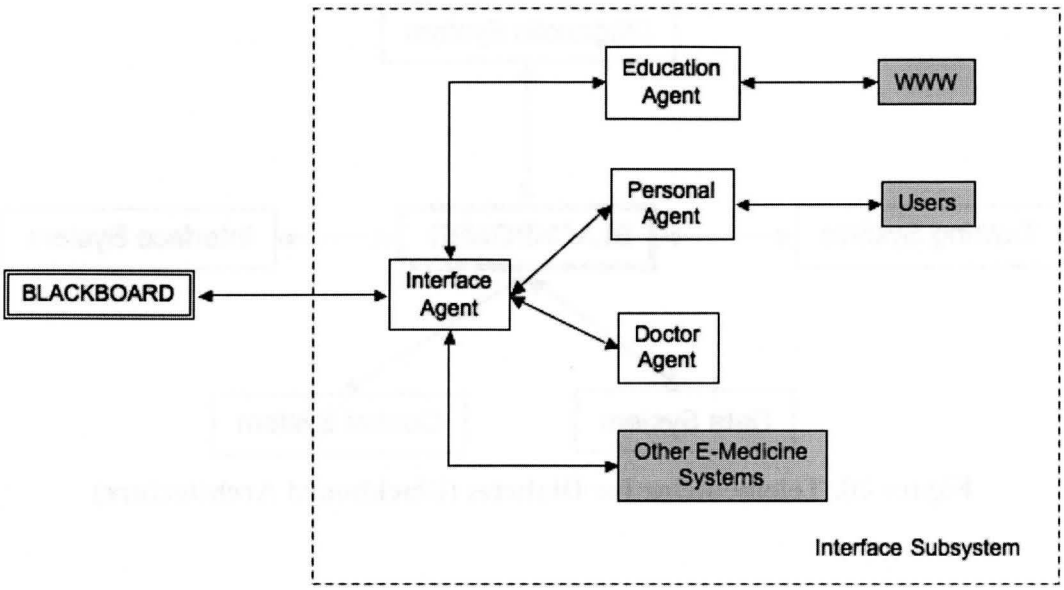

Figure 21. Interface Knowledge Source

- **Diagnostic**. It includes all the elements from the implementation group proposed in the original architecture, excluding the training, education and record agents. The goal of this knowledge source is to propose a therapy based on the information acquired from the blackboard through the monitoring, data processing, consultation and decision support agents. Once the information is processed, the results are sent to the diagnosis agents that make a suggestion for the patient specifying a therapy through the therapy agent.
- **Training**. This knowledge source just includes the training agent, which is based on the information from the diagnosis knowledge source. Its main function is to build a training program for the patient.
- **Database**. It has the database that stores all the information related with the patients. It is proposed as a different knowledge source because it manages all the information and the operations related with the database.
- **Control**. Composed of the Department, Administration and Controller Agents. Depending on the requests of the user, the department agents get information from external departments and send it to the administration agent, which assigns the tasks to the rest of the system. On the other hand, the controller manages the different knowledge sources and agents. All the elements of this group interact directly with the blackboard.

This system was clearly designed without using any architectural pattern; however, it was possible to analyze it from a layered and a data oriented point of view using a multi-layered and a blackboard architectural pattern, respectively. A

multi-layered style is less flexible; therefore, adaptation of new components its harder. However, if a correct abstraction level is used, it provides strong reusability and can be used as a base for further architecture.

The blackboard architecture is more flexible, and can be extended easily; if more systems are needed, they just need to be added to the blackboard and form part of the result. If another system must use this information, it just has to be connected to the blackboard. Both architectures are very helpful and, compared with the original design, they are clearer.

6.5. SIMPLE – A Multi-Agent System for Simultaneous and Related Auctions

6.5.1. Description

In Electronic Commerce, auctions are a way of negotiation that is getting a lot of attention. Compared with the traditional way of buying, they provide two main advantages: through them it is possible to achieve better prices supported by effective transactions and they integrate sellers and buyers from different parts of the world.

In ordinary auctions, items are sold in an isolated way. However, in order to satisfy customers' needs, a good idea is to sell related items at the same time and pack them together. A common example is a vacation package where it is possible to find optimal deals through the correct combination of flights and hotel nights.

The SIMPLE [37] agency is a trading Multi-Agent System using simultaneous auctions. It is based on the Trading Agent Competition (TAC) [58], which is an international forum that promotes research for the construction of trading agents. TAC's main objective is to minimize the cost of the packages in an electronic travel agency.

Travel packages contain three elements: a round-trip flight, a hotel reservation, and tickets for entertainment events. The traveler can choose the days for the trip or suggest a set of dates within a range of days. Additionally, they can have individual preferences, such as hotel type, flight class, kind of entertainment, and so on. Auctions are traded with different rules (a total of 28), prices can increase or decrease during the process, and auctions are updated every minute [49].

6.5.2. Initial Design

This system does not follow a specific architecture; the communication between the agents is direct. However, there is an agent managing each database and, according to the events in the environment and the databases, the agents react and execute different operations. The initial design is shows in Figure 22. The Market, Market KB and Solver are represented as a double lined box; the rest of the components are agents and are represented as a box. The agents that comprise the architecture are:

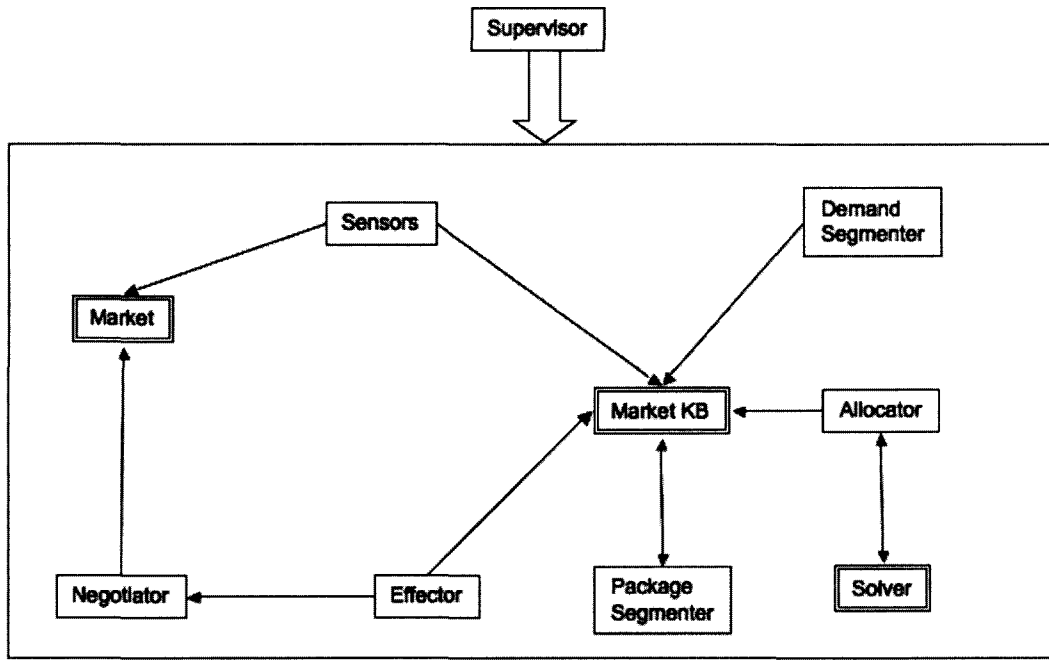

Figure 22. SIMPLE Architecture

- **Sensor Agents.** Responsible for collecting the data from the environment; they decide what data is relevant to the agents. They are reactive agents that just respond to events in the market; when some information on the auctions is generated, they transfer this information to the *Market Knowledge Base*.
- **Demand Segmenter.** Classifies the clients according to their characteristics and preferences. There are three types of costumers: *easy, medium and hard*; they are classified according to the flexibility on traveling dates. For example, if a costumer is flexible in his dates, or he is not going on critical nights, he is considered as easy.

- **Allocator.** Reactive agent that perceives changes from the Market Knowledge Base and performs communication with the Solver, which returns a set with the optimal set of goods that can be purchased.
- **Package Segmenter.** Separates the allocations depending on the score. This separation helps to identify goods that are common to the highest allocations.
- **Effector.** Calculates costs and sends them to the negotiator agents.
- **Supervisor.** Starts the system and initializes agents.
- **Negotiators**. Based on the information given by the effectors, they negotiate the costs trying to maximize performance and minimize cost. There are two types of negotiators: reactive and adaptive. The reactive ones start the game by bidding the minimum values; if it is necessary, they increment the bids and this process is repeated until the bid is accepted or the maximum value is reached. The adaptive ones increase the value following mathematical formulae and continue this process until the bid is accepted.

In addition, there are three components:

- **Market**. Environment that has all the information for the system.
- **Market Knowledge Base**. Internal representation of the data of the auctions. It stores the price quotes of each auction, goods already acquired by the system, client's preferences, closed auctions and information related with the agent's strategies.
- **Solver**. Module that assigns the possible costs to the packages. It receives data related with the clients from the Allocator, decides the number of goods that will be bought in the auction, and calculates the score for the optimal set of goods that will be purchased

6.5.3. Architecture Analysis

SIMPLE has two fundamental characteristics in its architectural design:

- Components react when new information arrives or changes happen in the Market Environment.
- All the received data and processed information is saved in a knowledge database, which is the central component.

When reactivity is added to components (in this case agents), it is possible to talk about an event-driven architecture. An event can be announced explicitly or implicitly. Explicitly means when a component needs to invoke a service defined in another component, it must initiate the invocation and wait for the result. Implicitly means when, instead of invoking a component directly, it can announce one or more events. The other components in the system register an interest for particular events; when they occur, they invoke the associated procedures. Depending on the information in the environment, an event is generated implicitly causing the invocation of procedures in other modules [15].

The *implicit invocation* pattern is a decentralized architecture that has components that signal events without a particular recipient. Elements communicate with each other when a registered process is triggered by an event [53]. In this case, sensors perceive all the changes occurring in the environment and send them to the database; however, they do not know directly which components will be affected by these events.

From another point of view, an important issue in this application is the understanding of the data in the system. For systems where data has a high degree of structure and where the execution order is determined by the incoming requests, [11] suggests using a database management system, which is considered as a repository. When systems react according to operations and changes occurring in a database, it is possible to talk about an *Active Repository* architectural pattern [3], which is an extension of a Repository Architecture.

In an Active Repository, clients are immediately notified of events in the database, such as changes or access to data. Notification mechanisms are realized in most cases by implicit invocations; however, it is possible to use explicit invocations. In SIMPLE, it is possible to identify an Active Repository pattern (Figure 23).

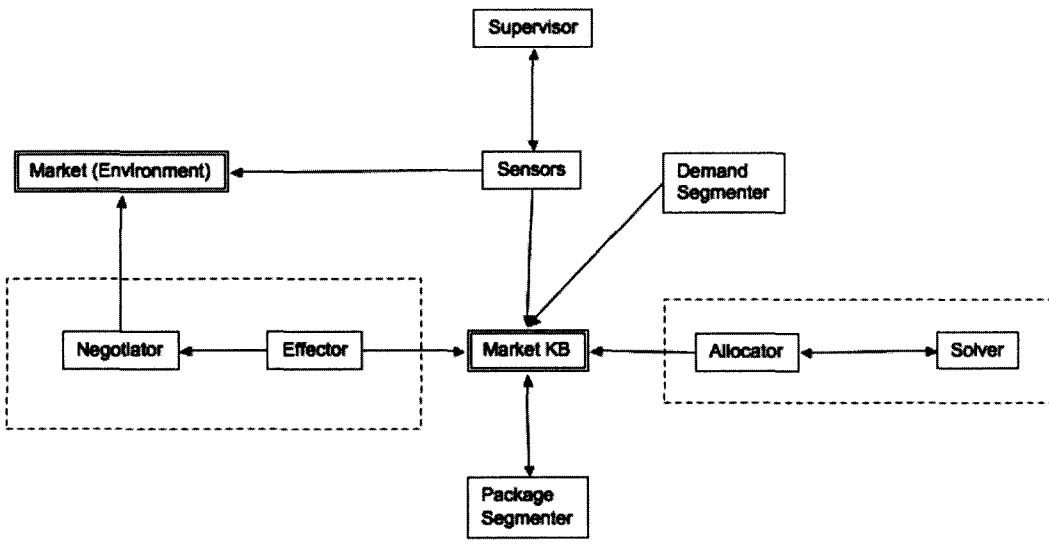

Figure 23. SIMPLE Reactive Repository Architecture

In simple, the components that compose the repository architecture are:

- **Repository.** Where all the information is saved. In SIMPLE, the Market Knowledge Database acts as a repository that is accessed by the effector,

sensors, demand segmenter, package segmenter and allocator agents. As part of the implicit invocation architecture, the component is also in charge of controlling events.

- **Negotiator and Effector Agent.** These agents can be grouped as a subsystem, the one that access directly to the repository is the effector; however, the negotiator is the one that interacts with the environment when there are changes on the database.
- **Allocator and Solver.** These agents are also grouped as a subsystem. The allocator interacts with the database and the solver supports for the execution of some operations.
- **Supervisor**. It is an independent agent that does not interact with the system during its execution; it just activates the subsystems. It can be connected with all the subsystems; however, in this case, it can activate the sensors, and depending on the changes that have occurred, sensors send events that activate the subsystems. This agent is extra to the reactive repository pattern.

In this case some agents were grouped in order to compose a subsystem; these groups are represented as a dashed box. The environment, which is represented as a double lined box, is not considered as an agent or subsystem, it is an information source.

Operations related with an user, like modifications in his/her profile and queries, generate events that activate the demand segmenter. Changes in the market knowledge database activate the allocator agent, which communicates with the solver; operations executed by the allocator, activate the package segmenter. Once a package is ready, the effector is activated and sends the information to the negotiator. This information is sent to the user and market, generating changes that are detected by the sensors; the process is repeated as necessary.

Even though SIMPLE was not originally designed using any architectural pattern, it was possible to identify a Reactive Repository Architecture, which has two main elements: events and a database. This architecture supports extendibility, when new subsystems are added and it is necessary to integrate new procedures in the subsystems that react to the new events. It also provides reusability; databases are one of the most used architectures, there is a lot of technology and references focused on them; therefore, all the advantages related with them can be applied.

6.6. Agent Based Simulation Architecture for Evaluating Operational Policies in Transshipping Containers.

6.6.1. Description

SimPort (Simulated container Port) is an Intelligent Decision Support System (IDSS) based on multi-agent technology that simulates the main transshipping operations in a container terminal and that assists the terminal managers to take decisions in the transshipping process. [21]. A c*ontainer terminal* is used to control operations for transporting goods. Its main functions are: to provide an optimal way to control the cargo, to enable ways to increase the transportation capacities, and to minimize the number of operations.

The transshipping operations in moving containers are divided into four processes:

- **Ship Arrival**. Allocates a berth position for the arriving ships and decides when it is possible to place them. This decision is based on sequence and positioning policies and acts as a main influence for the other operations.
- **Loading and Unloading Goods**. Allocates *quay cranes* and *straddle carriers* for each ship. It runs in parallel to the ship arrival process.
- **Horizontal Transport**. Sets the transportation in a continuous way reducing waiting time as much as possible. Additionally, it manages the load sequence, routing, pick up and coordination of quay cranes.
- **Yard Stack / Stack on Quay**. It stacks goods following stacking policies and properties such as stacking density, yard stack configuration, container allocation and dwell times.

6.6.2. Initial Design

The simulation technique used in SimPort is called MABS (Multi-Agent Based Simulation). It is applicable in distributed domains and it is used to study the interaction between the components that are part of a complex system. In this case, the following managers are modeled as agents: *port captain, ship agent, stevedore and terminal manager*. In addition, *quay cranes* and *straddle carriers* are also modeled as agents.

The system uses *reactive agents*, which make decisions based on the messages received. The behavior of the system depends on the behavior of each component when is placed in it is environment [21]. The architecture used in SimPort is presented in (Figure 24), and the agents that compose it are:

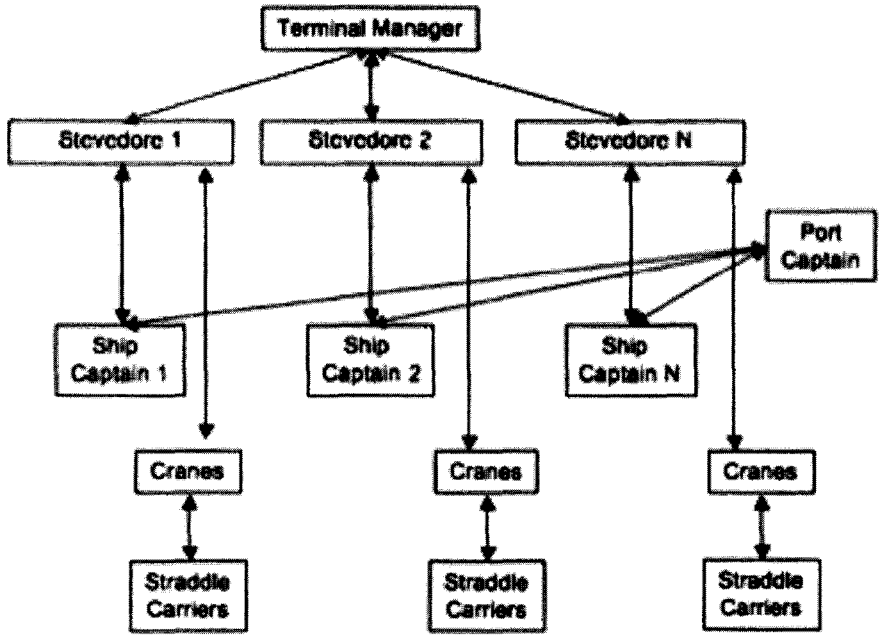

Figure 24. Simport Architecture

- **Port Captain Agent**. Sensor that is waiting all the time for arriving ships. Once a ship arrives, it creates a ship slot with information related with the ship's serving order and the sequence policies.
- **Ship Agent**. Each ship agent represents an arriving ship; they have information about the desired service time, length of the ship, position in the ship line, operating cost, etc.
- **Stevedore Agent**. Satisfies requests of the shipping agents such as more quay cranes. It also allocates the cranes provided by the terminal manager agent.
- **Terminal Manager Agent**. Allocates the berth position for a ship and allocates the cranes that will service a ship.
- **Quay Crane Agents**. Represents each quay crane that is coordinated by a stevedore agent during the execution of different operations. It loads and unloads containers as fast as possible and uses the straddle carrier agents to move the containers.
- **Straddle Carrier Agents.** Represents each straddle carrier and reacts to the requests from the Quay Crane Agents.

When a ship is arriving, it initiates communication with the port captain agent, which assigns a ship agent to it. When is necessary, a ship agent contacts the stevedore agent. This agent communicates with the terminal agent in order to

request a crane. Once that the assignment is done, the stevedore agent communicates with the crane agent that will make a request for straddle carrier agents. After that, the stevedore notifies to the terminal manager that the ship unloading is complete, so the ship can depart.

6.6.3. Architecture Analysis

SimPort does not follow any specific pattern, however, it can be structured from a multi-layered point of view into four layers (Figure 25):

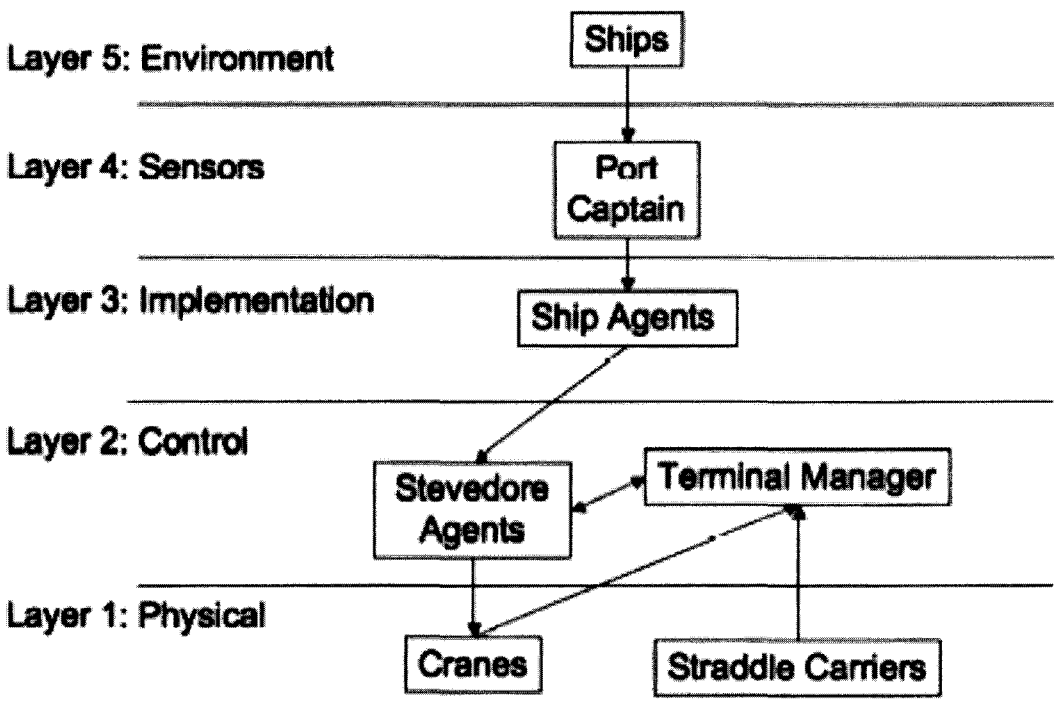

Figure 25. SimPort Multi-Layered Architecture

- **Layer 1: Physical.** Composed of Cranes and Straddle Carriers that are an agent representation of physical devices. They satisfy requests from the control layer.
- **Layer 2: Control.** This layer is in charge of coordinating and to satisfy the requests from the previous layers. It is composed of the Stevedore Agents and Terminal Managers that satisfy the requests of the next layer.
- **Layer 3: Implementation.** It is composed of agents that represent arriving ships. These agents use services from the previous layers and executed the necessary operations.

- **Layer 4: Sensors.** Composed of the Port Captain Agent, which acts as a sensor getting all the new arriving ships from the environment. Once the Port Captain detects an arriving ship, it creates a ship agent that will execute all the ship's requests.
- **Layer 5: Environment.** At the most abstract level, there are the arriving ships, which are detected by the Port Captain agent, which is placed at the previous level.

Moreover, the main characteristic of SimPort is that it is mainly based on reactive components and events. Agents request a service as an event and wait until it is satisfied. The events considered in the system are:

- A ship indicates that it is arriving.
- A ship agent requests a service.
- A stevedore agent requests a crane.
- A terminal manager agent requests a berth position.
- A quay crane requests a straddle carrier.

The reactor architectural pattern [31] permits event-based applications to separate services and to send them to the client that requested them. The architecture includes three main components: event sources, reactor and destination components. In a more detailed level, this pattern also proposes to include Handles, Synchronous Event Demultiplexer and Event Handlers; however, they are part of a more detailed level, that is, a design pattern. This system can be represented in terms of this architecture as follows (Figure 26):

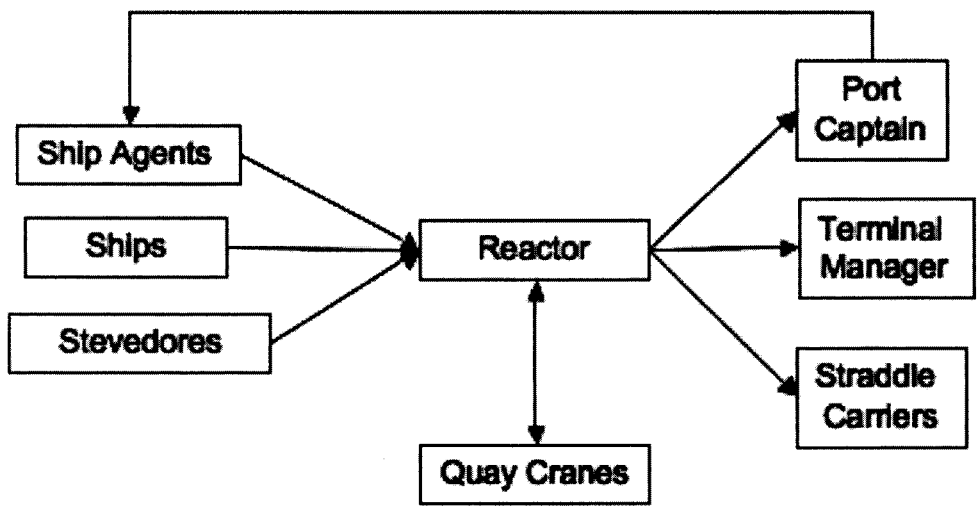

Figure 26. SimPort Reactor Architecture

- **Event Sources**. It is comprised of the ships, ship agents and stevedores. When it is necessary, they request for services notifying their events to the reactor.
- **Destination Components**. When an event occurs, these components are activated satisfying requests or executing operations as need. In this case, the destination components are the Port Captain, Terminal Agent, and Straddle Carrier.
- **Quay Cranes**. These agents generate events and also react with some of them. For a better design, it is better to split this component in two parts. One should correspond to the event sources, and the other one to the destination components. However, it is acceptable to have both situations in the same component.
- **Reactor**. Registers and controls event handlers. Once they are satisfied, it removes them from the list. When a service is available, and an event can be satisfied, the request is executed; therefore the event is committed. This component is an addition to the initial design.

7. Results

7.1. Research Results

- **There is an absence of software architectures for Multi-Agent Systems.** During the course of this research, six case studies were analyzed. We found that almost all of them have an initial design; however, these designs, even though they are composed of interconnected components, do not follow any pattern and they do not satisfy any structural rule. The only exception was MASACAD, which explicitly uses an architecture that is based on a variation of the broker pattern; however, this pattern is not exclusive to Multi-Agent Systems.

  In the Telemedicine for Diabetes case, they propose an "architecture" exclusively used for e-medicine; they present an initial design, which is extended when used for telemedicine in diabetes. However, aside from the fact that telemedicine is not a specific type of system, such as reactive systems, runtime system, and so on, it is not clear what are the components that compose the architecture and how it is structured; therefore, the proposed design it just a reduced version of a detailed design used for telemedicine.

- **It is possible to use existing patterns for building software architectures in Multi-Agent Systems**. In the literature, there are authors, such as Sylvain [50] and Hayden [20], who propose the use of existing patterns in Multi-Agent Systems. Following this idea, in many case studies, it was possible to restructure the initial design using existing patterns. In some cases, such as the Robot Disassembly Process, following the initial design, it was possible to find an implicit architecture based on existing patterns.

  Other cases, such as MASACAD, showed an explicit architecture based on existing patterns; however these patterns are not used exclusively in Multi-Agent Systems. Therefore, we can conclude that, in order to design the architecture of a Multi-Agent System, existing patterns can be used. In this thesis, the following patterns were used: Layers, Broker, Blackboard, Repository, Implicit Invocation, Reactive Repository and Reactor.

- **The properties of Multi-Agent Systems affect the way that an architecture is designed**. During the analysis stage of a system, we try to get all the functional/non-functional requirements and the general properties needed to design the system. These properties focus a system toward a specific paradigm that adds more characteristics to it. The design of a system architecture is mainly influenced by all of these properties together; when a system is based on agents, the main properties that affect the architecture are:

distributivity, cooperativity, and the properties related with Intelligent agents (reactivity, pro-activity, sociability).

Within the literature, there are many systems that can follow different patterns. According to the required properties of a system, a pattern can be imposed and design an architecture based on it. Avgeriou[3] proposes classifying the existing patterns according to their architectural view. Based on the Multi-Agent Systems and Intelligent Agents properties, Multi-Agent System architecture can be classified following four points of view: *layered view, data-centered view, component interaction view* and *distribution view.*

Because Multi-Agent Systems are distributed, according to the functionally of the agents, it is possible to group the agents into subsystems. According to the services that they provide, the components can also be grouped into layers. Therefore, architectures such as Layers and Broker, are very well suited for Multi-Agent Systems.

In addition, the way that agents communicate between each other is another important property that affects the way that an architecture is designed. In Multi-Agent Systems, agents can communicate through a central component that provides services and coordinates the subsystems. They also can communicate through a shared component where the components save the information. Therefore, according to the communication between agents, a broker, blackboard, and repository architectures are very well suited for Multi-Agent Systems.

Finally, in a Multi-Agent System, the agents react according to the changes that occur in the environment; this property implies the use of events. When a system is based on events, it is possible to use architectures such as implicit invocation, active repository, and reactor. An architecture can be designed using different points of view; therefore, a system is not restricted to follow architectures based just on one pattern.

- **The initial design of many systems can be reengineered, and as a result, propose architectures based on existing patterns**. As we stated before, even though there are not specific architectures for Multi-Agent Systems, some systems show explicit architectures based on existing patterns. Although most of the analyzed systems did not show a software architecture; it was possible to re-design the initial design and propose an architecture following an existing pattern.

  In this research, all of the re-engineered architectures followed an existing pattern; however, we cannot discard the possibility of finding architectures unique to MASs through the analysis of more systems. As another result we

can conclude that, even though usually Multi-Agent Systems do not use the software architecture concept, designers use it without knowing; therefore, a software architecture can be found in an implicit way.

- **There are different uses of the "software architecture" term**. In the literature, many authors, for example Flores-Mendez [14], use the term architecture referring to abstract architectures proposed in standards, such as FIPA and OMG. Abstract architectures define the elements of a system and their organization in an abstract way. For example, FIPA defines that a Multi-Agent System must have an agent server, a services server, yellow pages and a communication language; in order to find and agent or a service, the yellow pages and white pages must be accessed. However, it does not specify which agents are indispensable, how they communicate with each other, and so on.

  Additionally, usually an initial design for a system is presented; in almost all cases, authors refer to this design using the "software architecture" term. The way that it is used does not correspond to the definition used in this research and in the general literature on Software Architectures. Although initial designs present interconnected components, they do not follow a pattern and they do not satisfy explicit structuring rules.

- **Standards and Frameworks do not affect an architecture design**. A standard provides a set of guidelines that help build a better implementation of a design; however, they do not affect the architecture; they just act as a reference for how components must be composed in a system at an implementation level. For example, MASEL was implemented following the FIPA standard, which provides an abstract architecture that indicates the components that the application must have at an implementation level. MASEL was developed using JADE, which is a framework that provides a set of tools for an easier implementation. MASEL follows two main architectures: repository and layers (three tier); however, both architectures just represent the structure of the system. Both architectures do not specify the communication language that is used, their location, how they can be accessed, and how a message can be specified. For these purposes, it is possible to use the standards. Finally, once all these properties are specified, JADE is used to implement the application in Java.

- **Methodologies influence the way that an architecture can be designed**. Because the scope of this research, methodologies in Multi-Agent Systems were not analyzed in detail. However, through the given overview we noticed that methodologies do not affect the design of an architecture; however, they help to analyze the system at an early stage.

Methodologies provide guidelines for modeling the requirements and components in a way that allows an architectural pattern to be chosen intuitively. As a result of the use of methodologies, an architecture can be designed from different points of view, and as a consequence, the design can take different directions and can be based on different patterns.

7.2. Case Studies Comparison

During this research, different case studies were analyzed in order to study their software architecture. As was discussed in the previous section, we found that in all the cases, there was not an explicit architecture for Multi-Agent System. However, most of them showed an architecture specifically designed for each system.

As a commonality, it was possible to redesign all architectures using existing architectural patterns. Analyzing all the systems, we found that, according to their properties, it is possible to reuse the same pattern in different cases. As a result of the similarities found and the redesign using existing patterns, it was possible to compare similar cases and get some conclusions. This comparison is detailed in the following points:

- **MASEL, Telemedicine for Diabetes and SIMPLE.** All share a main characteristic: they are data-centered. MASEL and SIMPLE have a central component, a database; however, in the first one, clients do not access it directly. In addition, there is an intermediate layer that controls processes and data; therefore, as a consequence, it follows a three-tier architecture.

  In SIMPLE, components are grouped in subsystems that access the database directly; therefore, it is not necessary to use a layered style. Additionally, the system is based on events, which suggests adding reactivity properties to the database, i.e., an event can be triggered whenever information is updated, inserted, or deleted.

  Even though Telemedicine for Diabetes has a data-oriented view, it does not use a database. It has subsystems that solve a part of the problem and that are coordinated by a control component. The partial solutions are stored in a blackboard component and all together compose the solution for the problem. Because of the complexity of the application, and the way that the data is modeled; this system can also be modeled according to the components that communicate through a layered architecture that is structured in a hierarchical way.

- **Robot Disassembly Unit and MASACAD.** Both are systems with distributed components; however, the design style of each one is distinct. In the first one,

one system executes operations that produce results that are sent to the next system. These results are processed and the resulting information acts as an input to the third system, which is composed of physical robots. In this case, it is possible to decompose the system according to the tasks assigned to each agent, and structure it in a hierarchical way using layers.

MASACAD can also be modeled in a distributed way; however, in contrast with the previous case, the communication between them is not sequential. In this case, the broker architecture is the best-suited architecture; therefore, all the subsystems are controlled by a common component. The broker architecture is also based on a multi-layered style that decomposes the system in a natural way. It is important to remark that in this case the system does not share a common data repository. In addition, there is not a control component that decides the activation and execution of the system and the components are independent; therefore, it is not possible to follow a blackboard architecture.

- **SIMPLE and the Transshipping Containers System**. They share something in common: both are event-based systems; therefore, both are based on an implicit invocation architecture. This type of architecture does not have a complex structure, it is mainly composed of components and events; however, if these components are added to a repository, it is possible to use an active repository pattern. On the other hand, the Transshipping Containers System presents a more specific type of event-based architecture: the reactor. This one is more detailed and provides more control for the system. In this case, it is possible to control all the events in a centralized way through the reactor, which is the main component.

As a result of the analysis of these systems, we can conclude that:

- Data-repositories are very helpful when it is necessary to store common information in a shared component, such as a database.
- If the communication consists of several intermediate steps, a good option is to use a layered architecture.
- If the system can be decomposed in subsystems that can solve a part of the problem, blackboard architectures present an optimal approach.
- When data changes affect the behavior of the system, an active repository can be used.
- Layers can be used to model the communication between components when they can be modeled in a hierarchical way.
- A distributed design that is composed of many subsystems can be modeled using a broker architecture; the interaction between the components will be coordinated by the central component: the broker.

- When the application is event-oriented, an implicit event architecture can be used, but it does not specify exactly the components that must be included in the design.
- If in an event-based architecture, all the events can be controlled by a central component, and each component has a method that reacts according to the trigged action, it is possible to use a reactor architecture.

7.3. Case Studies Summary

Table 1 and 2 summarizes the main properties of each case study, and also it presents the patterns used and the results obtained:

	Robot Disassembly Process	MASACAD	MASEL
Original Architecture	--	Bee-gent	--
Main Properties	Distributed, decomposed into subsystems, interacts with physical agents	Distributed, composed of subsystems, uses two data-bases, meditation agent	Data-centered, composed of three databases, clients ask for services, semi-distributed
Views	Layered	Layered, distribution	Data-centered/layers
Proposed Pattern	Layers	Layers, Broker	Layers (3-their), repository
Problems Found	Unclear design, no explanation about external systems, confusing separation between subsystems		Redundant interaction between components, many intermediate components for accessing the database, databases managers provides services unrelated with the database, unclear separation between the distribution of components
Benefits	Graphical architectural representation, hierarchical interaction of components, separation of physical elements, easy to see the functions in each layer, easy to add new systems	Clear distinction between clients and servers, easy integration of systems, easier control of the system, clear separation of tasks.	Integration of all databases in one component, simplified distribution of components, elimination of redundant communication, easy to add new systems, separation client – server, database managers can be incorporated to the database component.

Table 1

	TELEMEDICINE	SIMPLE	Transshipping Containers
Original Architecture	--	--	--
Main Properties	Subgroups of components, partial solutions	Data centered, event-based, reactive agents, elements are executed implicitly	It includes physical devices, system controlled by a supervisor, event-based, events are coordinated in common components
Views	Data-centered, layered	Data/centered, component interaction	Component interaction
Proposed Pattern	Layers, broker	Reactive repository	Layers, reactor
Problems Found	Unclear description of many components, groups not very well defined, unorganized and unclear interactions, complex communications	Supervisor interacts with all the agents, besides the interaction with the database, subgroups interact with each other, not all components are modeled as agents	Unclear control system, architecture can be confused with a blackboard, not clear the focus of the application.
Benefits	Integration of all the databases in the same component, simplified distributivity, elimination of redundant interactions, system can be added easily, agents can be grouped according to their function, each subsystem can be designed in a different way.	Simplification of communication between components, it possible to group components and model them as a subsystem, easy addition of systems, events can be coordinated in the database, each subsystem is independent.	Clearer interaction between components, agents of the same type can be considered as a unit, events are controlled in one component, and the abstraction level is smooth between components.

Table 2

8. Conclusions and Future Work

8.1. Conclusions

One of the main goals of this thesis was to analyze different case studies in order to see if they make explicit use of software architectures and whether these software architectures are specific for Multi-Agent Systems. After analyzing a number of Multi-Agent Systems applications and researching the literature, we found that there do not exist architectures that are exclusive to Multi-Agent Systems.

Furthermore, we found that, within the literature, the term *software architecture* is used in many ways. One of them refers to reference architectures proposed in standards, such as FIPA and OMG. In other cases, such as Cougaar, authors refer to frameworks as a software architecture; however, they are just a set of tools that help for an easier deployment of software architectures; that is they are examples of middleware.

Additionally, analyzing the case studies, we could see that they never use explicit architectures for Multi-Agent Systems; however, they use an initial design for each case. In many cases, this design is called "architecture"; however, it does not correspond to the software architecture used in this thesis. Moreover, based on the requirements of each case and the properties provided by Multi-Agent System, it was possible use existing patterns for re-designing the original architecture.

It was also possible to compare the cases searching for similarities. According to their commonalities, we found that it is possible to use similar architectures in different cases. Finally, summarizing the results found, we have that:

- There is an absence of software architectures for Multi-Agent Systems.
- It is possible to use existing patterns for building software architectures in Multi-Agent Systems.
- According to the properties of Multi-Agent Systems, such as distributibity, cooperativy and reactivity, it is possible to design architectures following existing patterns, such as broker, blackboard, layers and implicit invocation, which are very well suited for Multi-Agent Systems.
- Many applications have an initial design, which is composed of components and connectors. However, it does not follow any pattern; therefore, it cannot be considered as a software architecture.
- In many systems it was possible to recognize existing patterns and propose software architectures that are based on them.
- Methodologies suggest the way that an architecture can be used.

- Different applications share many characteristics; based on them, it is possible to reuse existing patterns for similar cases.

8.2. Future Work

- **Software Architecture for Multi-Agent Systems**. Currently there are no software architectures that are specific for Multi-Agent Systems; within this kind of system, there is a big variety of applications, therefore, it is hard to suggest one architecture for all the Multi-Agent Systems. However, there is the possibility to propose software architectures for different subsets of Multi-Agent Systems.

- **Use of more architectural patterns**. The patterns analyzed in this research, are based on some case studies; however, there are more systems with different properties that can use patterns that were not studied in this thesis.

- **System implementation using proposed architectural designs**. All the systems were implemented using an initial design; however, none of them have been implemented using the proposed architectures based on existing patterns. A very interesting idea is to implement them, and compare them analyzing the differences.

- **Use of architecture of other type of systems**. Multi-Agent System have some similarities with different types of systems, such as reactive systems, real time systems, and so on. Another interesting research topic is to see if it is possible to borrow styles from different domains and apply them to Multi-Agent Systems.

- **Design patterns for Multi-Agent Systems**. This work is focused on software architectures; however, in a more detailed level, there are existing designs patterns that can be used for Multi-Agents Systems at an implementation level. This thesis does not address anything related with design patterns.

9. References

[1] M. T. a. T. W. Aaron Helsginer, *Cougaar : A scalable, distributed multi-agent architecture*, International Conference on Systems, Man and Cybernetics, 7 (2004), pp. 123-134.

[2] ALPINE, *Cougaar Architecture Document*, BBN Technologies, 2004.

[3] P. Avgeriou, *Architectural patterns revisited — a pattern language*, Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005) (2005), pp. 1 -- 39.

[4] L. Bass, P. Clements and R. Kazman, *Software Engineering in Practice*, Addison-Wesley, Boston, MA, 2003.

[5] C. Baumer, M. Breugst, S. Choy and T. Magedanz, *Grasshopper: a universal agent platform based on OMG MASIF and FIPA standards*, 2000.

[6] F. Bellifemine, A. Poggi and G. Rimassa, *Developing Multi-agent Systems with JADE, ATAL '00: Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages*, Springer-Verlag, London, UK, 2001, pp. 89--103.

[7] F. Bellifemine, A. Poggi and G. Rimassa, *JADE - A White Paper*, EXP In search of innovation, 3 (2003).

[8] F. Bergenti, M.-P. Gleizes and F. Zambonelli, *Methodologies and software engineering for agent systems : the agent-oriented software engineering handbook*, Kluwer Academic, Boston [Mass.] ; London, 2004.

[9] L. Blass, P. Clements and R. Kazman, *Software Engineering in Practice*, Addison-Wesley, Boston, MA, 2003.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture, A system of Patterns*, John Wiley & Sons Ltd., West sussex, England, 1996.

[11] M. S. a. P. Clements, *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*, Proc. COMPSAC97, 21st Int'l Computer Software and Applications Conference (1996), pp. 6-13.

[12] CougaarForge, *Cougaar Website*, http://www.cougaar.org/, 2007.

[13] M. F. W. a. S. DeLoach, *An Overview of the Multiagent Systems Engineering Methodology*, First international workshop, AOSE 2000 on Agent-oriented software engineering (2000), pp. 207-221.

[14] R. Flores-Mendez, *Towards the Standardization of Multi Agent Systems Architectures: An Overview*, 1999.

[15] D. Garlan and M. Shaw, *An Introduction to Software Architecture*, in V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, Singapore, 1993, pp. 1-39.

[16] A. Garro and L. Palopoli, *An XML Multi-Agent System for e-Learning and Skill Management*, (2002).

[17] A. R. Hajo, *Design and Control of Workflow Processes*, Springer-Verlag New York, Inc., 2003.

[18] M. S. Hamdi, *MASACAD: A Multiagent-Based Approach to Information Customization*, IEEE Computer Society, Los Alamitos, CA, USA, 2006, pp. 60-67.

[19] Q. Hao and W. S. Z. Zhang, *An autonomous agent development environment for engineering applications*, Advanced Engineering Informatics, 19 (2005), pp. 123-134.

[20] S. Hayden, C. Carrick and Q. Yang, *Architectural Design Patterns for Multi-Agent Coordination*, 1999.

[21] L. Henesey, P. Davidsson and J. A. Persson, *Agent Based Simulation Architecture for Evaluating Operational Policies in Transshipping Containers, MATES*, 2006, pp. 73-85.

[22] M. N. Huhns and L. M. Stephens, *Multiagent Systems and Societies of Agents*, in G. Weiss, ed., *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, Cambridge, MA, USA, 1999, pp. 80--120.

[23] J. H. Ian Gorton, David McGee, Andrew Cowell, Olga Kuchar and Judi Thomson, *Evaluating Agent Architectures: Cougaar, Aglets and AAA, Software Engineering for Multi-Agent Systems II*, SpringerLink, WA, USA, 2004, pp. 264--278.

[24] IEEE-Computer-Society, *FIPA Abstract Architecture Specification*, 2001.

[25] IEEE-Computer-Society, *The Foundation for Intelligent Physical Agents Website*, http://www.fipa.org/, 2007.

[26] Jade-Project-Team, *Java Agent DEvelopment Framework Website*, http://jade.cselt.it/, 2007.

[27] N. R. Jennings, K. Sycara and M. Wooldridge, *A Roadmap of Agent Research and Development*, Journal of Autonomous Agents and Multi-Agent Systems, 1 (1998), pp. 7--38.

[28] N. R. a. W. Jennings, Michael, *Agent-Oriented Software Engineering*, in F. J. Garijo and M. Boman, eds., *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering ({MAAMAW}-99)*, Springer-Verlag: Heidelberg, Germany, 1999, pp. 1--7.

[29] T. Kawamura, T. Hasegawa, A. Ohsuga and S. Honiden, *Bee-gent: Bonding and Encapsulation Enhancement Agent framework fordevelopment of distributed systems*, Software Engineering Conference, 1999. (APSEC '99) Proceedings. Asia Pacific, 6 (1999), pp. 260-267.

[30] M. W. a. N. R. J. a. D. Kinny, *The Gaia Methodology for Agent-Oriented Analysis and Design*, Autonomous Agents and Multi-Agent Systems, 3 (2000), pp. 285--312.

[31] M. Kirchner and P. Jain, *Pattern-oriented software architecture. Volume 3*

patterns for resource management, Wiley, West Sussex, Eng. ; Hoboken, N.J., 2004.

[32] H. N. a. D. N. a. L. Lee, *ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems*, In Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems (1998), pp. 377-392.

[33] M. MacKenzie, K. Laskey, F. McCabe, F. Limited, P. Brown, R. Metz and B. A. Hamilton, *Reference model for service oriented architectures*, *OASIS SOA Reference Model*, OASIS OPEN, 2006, pp. 1-28.

[34] P. C. Mary Shaw, *The Golden Age of Software Architecture*, IEEE Software, 23 (2006), pp. 31-39.

[35] O. MASIF, *Mobile Agent Facility specification*, (2000).

[36] B. M. Michelson, *Event-Driven Architecture Overview*, Patricia Seybold Group (2006).

[37] R. L. Milidiu, T. Melcop, F. dos S. Liporace and C. J. P. de Lu, *SIMPLE - A Multi-Agent System for Simultaneous and Related Auctions*, *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, IEEE Computer Society, Washington, DC, USA, 2003, pp. 511.

[38] P. J. Modi, S. Mancoridis, W. M. Mongan, W. Regli and I. Mayk, *Towards a reference model for agent-based systems*, *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM Press, New York, NY, USA, 2006, pp. 1475--1482.

[39] T. Norman, N. Jennings, P. Faratin and A. Mamdani, *Designing and Implementing a Multi-Agent Architecture for Business Process Management*, in l. J. o. r. P. M\"u, M. J. Wooldridge and N. R. Jennings, eds., *Proceedings of the {ECAI}'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents {III}*, Springer-Verlag: Heidelberg, Germany, 1997, pp. 261--276.

[40] M. Nowostawski, G. Bush, M. Purvis and S. Cranefield, *Platforms for agent-oriented software engineering*, apsec, 00 (2000), pp. 480.

[41] OMG, *Object Management Group Website*, (2007).

[42] M. Oprea, *Applications of Multi-Agent Systems*, IFIP International Federation for Information Processing, 1 (2004), pp. 239--270.

[43] A. Pavliska and V. Srovnal, *Robot Disassembly Process Using Multi-agent System*, *CEEMAS '01: Revised Papers from the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems*, Springer-Verlag, London, UK, 2002, pp. 227--233.

[44] D. E. Perry and A. L. Wolf, *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes, 17 (1992), pp. 40--52.

[45] C. J. Petrie, *Agent-Based Software Engineering*, in J. Bradshaw and G. Arnold, eds., *Proceedings of the 5th International Conference on the*

Practical Application of Intelligent Agents and Multi-Agent Technology ({PAAM} 2000), The Practical Application Company Ltd., Manchester, UK, 2000.

[46] P. G. a. M. K. a. J. M. a. M. Pistore, *The Tropos Methodology: an overview*, Methodologies And Software Engineering For Agent Systems (2004).

[47] S. Poslad and P. Charlton, *Standardizing agent interoperability: the FIPA approach*, (2001), pp. 98--117.

[48] P. Reed, *Refernce Architecture: The best of best practices.*, *DeveloperWorks - Rational*, 2002.

[49] Ruy L. Milidiu, Taciana Melcop, Frederico dos S. Liporace and C. J. P. d. Lucena, *Multi-Agent Strategy for simultaneous and Related Auctions*, Scientia, 14 (2003), pp. 155-170.

[50] S. Sauvage, *MAS Oriented Patterns*, Lecture Notes In Computer Science;, 2296 (2001), pp. 283-292.

[51] D. C. Schmidt, *Pattern-oriented software architecture: Patterns for concurrent and networked objects, volume 2*, Wiley, New York, 2000.

[52] M. Shaw, *Some patterns for software architectures*, In Proceedings of the Second Pattern Languages of Program Design Workshop (1996).

[53] M. Shaw and D. Garlan, *Software architecture : perspectives on an emerging discipline*, Prentice Hall, Upper Saddle River, N.J., 1996.

[54] I. Sommerville, *Software Engineering*, Addison-Wesley, 2001.

[55] N. G. T. and et al., *Agent Platform Evaluation and Comparison*, Pellucid, Bratislava, Slovakia, 2002.

[56] J. Tian and H. Tianfield, *A Multi-agent Approach to the Design of an E-medicine System*, *MATES*, 2003, pp. 85-94.

[57] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*, MIT Press, 2000.

[58] M. Wellman, *Trading Agent Competition*, 2007.

[59] M. Wooldridge, *Agents and software engineering*, AI*IA Notizie, XI (1998), pp. 31--37.

[60] M. Wooldridge, *Intelligent Agents*, in G. Weiss, ed., *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, Cambridge, MA, USA, 1999, pp. 27--78.

[61] M. J. Wooldridge, *An introduction to multiagent systems*, J. Wiley, New York, 2002.

[62] E. Yu, *Agent Orientation as a Modelling Paradigm*, Wirtschaftsinformatik, 43 (2002), pp. 123--132.

[63] U. Zdun, M. Kircher and M. Volter, *Remoting patterns: design reuse of distributed object middleware solutions*, Internet Computing, IEEE, 8 (2004), pp. 60--68.