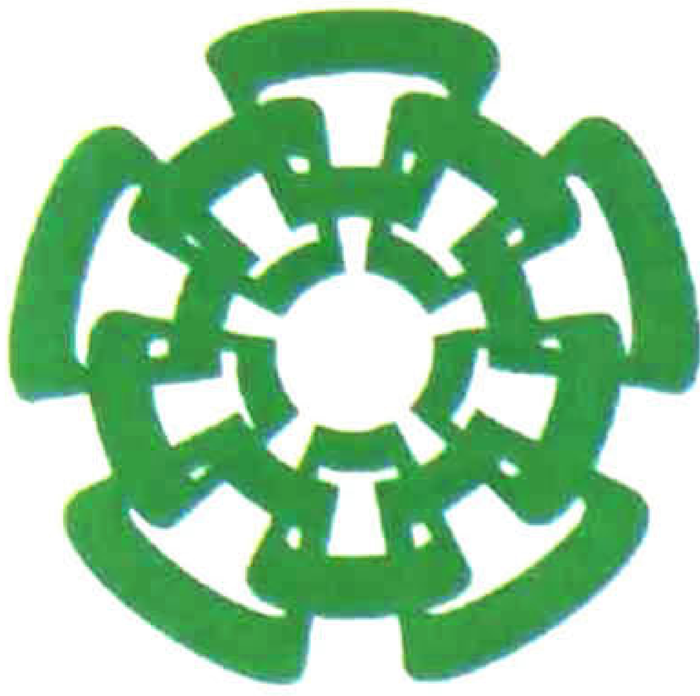
BC-652
Con 2011

xx(179000.1)

CINVESTAV
IPN
ADQUISICION
DE LIBROS



Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

Middleware para Distribución Adaptativa de Ambientes Multiagente - Middleware for Adaptive Multiagent Environment Distribution

Tesis que presenta:
Luis Alberto Muñoz Gómez

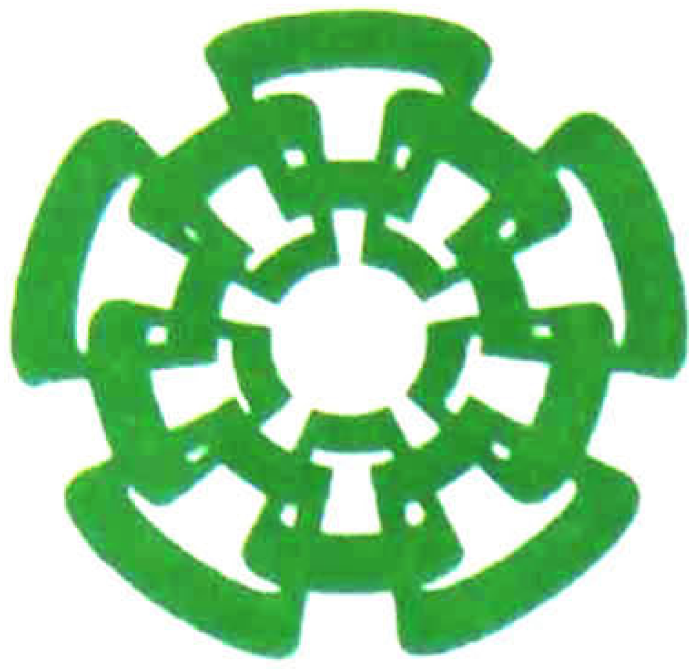
para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Director de Tesis
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Noviembre de 2010.

CLASIF.: TK165.G8 M86 2010
ADQUIS.: 351652
FECHA: 18-Agosto-2011
PROCED.: Don. 2011
ID: 174543-1001



Centro de Investigación y de Estudios Avanzados

del Instituto Politécnico Nacional

Unidad Guadalajara

Middleware for Adaptive Multiagent Environment Distribution

A thesis presented by:
Luis Alberto Muñoz Gómez

to obtain the degree of:
Master of Science

in the subject of:
Electrical Engineering

Thesis Advisor:
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, November 2010.

Middleware para Distribución Adaptatativa de Ambientes Multiagente - Middleware for Adaptive Multiagent Environment Distribution

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
Luis Alberto Muñoz Gómez
Ingeniero en Computación
Universidad de Guadalajara 1998-2002

Becario de CONACYT, expediente no. 199554

Director de Tesis
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, Noviembre de 2010.

Middleware for Adaptive Multiagent Environment Distribution

Master of Science Thesis
In Electrical Engineering

By:
Luis Alberto Muñoz Gómez
Computer Engineer
Universidad de Guadalajara 1998-2002

Scholarship granted by CONACYT, No. 199554

Thesis Advisor:
Dr. Félix Francisco Ramos Corchado

CINVESTAV del IPN Unidad Guadalajara, November, 2010.

Resumen

La realidad virtual representa diferentes entidades que interactúan en un ambiente para realizar alguna tarea. Cuando se implantan ambientes complejos, se necesita distribuir el procesamiento requerido para obtener la calidad deseada, lo mismo sucede cuando las aplicaciones son inherentemente distribuidas. En nuestro enfoque el comportamiento de cada una de estas entidades es calculada por un agente. Los agentes se comunican por medio de un middleware con varios componentes. Dependiendo del tipo de aplicación que se representa, algunos de los componentes pueden convertirse en cuello de botella, principalmente por la comunicación generada entre los agentes. En esta tesis se propone una solución para resolver el problema del cuello de botella al que tiende a convertirse la entidad que representa al ambiente. En nuestro enfoque, esta entidad se compone de un contexto que contiene información acerca del ambiente y un escenario que es el medio donde se desarrolla la escena. Durante el desarrollo de una escena, los agentes se desenvuelven acorde a sus percepciones acerca del ambiente, objetos y otros agentes a su alrededor. Esta entidad se convierte en cuello de botella porque cada agente consulta al ambiente para percibir de él, y también para notificarle cualquier intención de realizar modificaciones en su estado así como validar en el contexto las reglas para los cambios permitidos. Esta continua comunicación de todos los agentes con el ambiente provoca que el agente que controla al ambiente se transforme en un cuello de botella en el sistema y, por consiguiente, los otros agentes no puedan percibir eficazmente los cambios.

La solución propuesta en este trabajo consiste en distribuir adaptivamente el ambiente en un conjunto de agentes ambiente. Cada uno de los agentes ambiente representa un volumen x del ambiente y todos ellos son percibidos como una sola entidad ante los demás agentes. Decimos que la distribución es adaptativa porque el número de agentes que representan al ambiente es calculado en función de la complejidad del ambiente para cada tiempo t. Si el ambiente es complejo, entonces tendremos que el número de agentes ambiente es grande, de lo contrario es reducido. Además, la distribución se hace en función de la complejidad de cada región del ambiente, con lo cual tendremos más agentes ambiente resolviendo el cuello de botella en secciones del ambiente que contengan mayor complejidad y un menor número en las secciones con poca complejidad. Cuando un agente ambiente decide aplicar una dicotomía porque la complejidad aumenta, nuestra solución resuelve el problema de decidir donde generar un nuevo agente ambiente y qué información proporcionarle; de forma similar, si la complejidad disminuye, dos o más agentes ambiente acordarán como fusionar su información. Dicha solución es integrada como dos módulos de la arquitectura del proyecto GeDA-3D. Los módulos están constituidos por un middleware con funciones de soporte para hilos y comunicación confiable y, un conjunto de clases ambiente genéricas escritas en lenguaje Java. La decisión de usar Java además de ser un lenguaje multiplataforma es porque el lenguaje permite especificaciones heredables, y por tanto, reutilizables para cualquier ambiente específico que se desee poner en escena sobre la plataforma GeDA-3D.

Abstract

Virtual reality represents different entities which interact into an environment to carry out any task. When complex environments are implemented, we need to distribute required processing to obtain the desired quality, the same happens when applications are inherently distributed. In our approach an agent calculates each of these entities behavior. Agents communicate with each other through a middleware with several components. Depending of the kind of virtual reality that is represented, some components may turn into a bottleneck, mainly due to the communication generated among agents. In this thesis a solution is proposed to solve the problem of the bottleneck to which the entity that represents the environment tends to turn into. In our approach, this entity is composed by a context which contains information about the environment and a scenario that is the medium for a scene development. During a scene development, agents evolve according to their perceptions about the environment, objects and other agents around them. This entity turns into a bottleneck because each agent consults the environment to perceive from it, and also to notify it any intentions to carry out modifications to its state as well as to validate in the context rules for permitted changes. This continuous communication of all agents to the environment causes that the agent that controls the environment turns into a bottleneck in the system and, therefore, the other agents can not perceive changes effectively.

The proposed solution in this work consists in distributing adaptively the environment on a set of environment agents. Each of these environment agents represents a volume x of the environment and all of them are perceived as a single entity by other agents. We say that distribution is adaptive because the number of agents that represent the environment is calculated in function of the complexity of the environment for each time t. If the environment is complex, then we will have that the number of environment agents is big, otherwise is small. Besides, distribution is done in function of the complexity of each environment region, with which we will have more environment agents resolving the bottleneck in sections of the environment that contain more complexity and a smaller number in sections with low complexity. When an environment agent decides to apply a dichotomy, this decision is based on an important increase of complexity it manages. Our solution resolves the problem of deciding where to generate a new environment agent and what information is provided to it; in a similar way, if complexity diminishes, two or more environment agents will agree how to fuse their information. This solution is integrated as two modules of the GeDA-3D project architecture. These modules are constituted by a middleware with support functions for threads and reliable communication and, a set of generic environment classes written in Java language. The decision of using Java besides being a multiplatform language is because the language lets inheritable specifications and, therefore, reusable for any kind of specific environment that is desired to put in scene on the GeDA-3D platform.

Acknowledgements

To God for guiding me throughout my life, so curious, every step I go.

To my mother Francisca for being the best mother in the world.

To my grandmother Agustina for being the main pillar of my family and like a second mother to me all her life.

To my father Antonio for all his support and encouragement to overcome.

To my advisor Dr. Félix for believing in me, guiding me on this work and giving me every opportunity so far.

To my little sister Tania for becoming the spiritual pillar of my family in the most difficult moments.

To my little brother Alex for all his need of attention that occasionally took me from my work and brought out my inner child.

To my aunt Ramona for her encouragement.

To my best friends Salvador, Rosalva, Mary, Sandra and Jose Luis for all special moments that we have shared and that give sense to life outside home and work.

To my teachers from CINVESTAV (Doctors Ernesto López, Mario Siller, Raul González and Héctor Durán), who taught me interesting topics, some of which I applied on this thesis.

To my teachers from University of Guadalajara (Aarón Jiménez, Luis Casillas, Abelardo Gómez, Angélica Ancona, Manuel Corona and Salomón Ibarra), who taught me interesting topics that profiled me to my M.Sc. in Engineering.

To my friends from CINVESTAV and from University of Guadalajara for all convivial moments, from a simple nice conversation to parties and celebrations.

To my consented alumni of my Distributed Systems Workshop at the University of Guadalajara for contributing to my desire in continuing being a member of the academy.

To CINVESTAV for admitting me as a member of its community.

To CONACYT for its support granting me the scholarship no. 199554.

Contents

Chapter 1 Introduction 1
1.1 Problem Description 1
1.2 Goals 3
1.3 Proposal 3
1.4 Structure of this Thesis 5
Chapter 2 Review of Related Works 7
2.1 Introduction 7
2.2 Distributed Systems 7
2.2.1 Middleware 8
2.2.2 Load Sharing and Load Balancing 9
2.2.3 Replication and Mobility 11
2.2.4 Code Migration 13
2.2.5 Distributed Shared Memory 14
2.3 Multiagent Systems 16
2.3.1 Distributed Virtual Environments 17
2.3.2 Distributed Virtual Reality Platforms 18
2.3.3 Self Organization of Multiagent Systems 20
2.4 Software Engineering 22
2.4.1 Reverse Engineering 22
2.4.2 Component Based Software Engineering 23
2.5 Summary 26
Chapter 3 Software Engineering of GeDA-3D 29
3.1 Introduction 29

3.2 Redocumentation of GeDA-3D 31
3.2.1 The Context Module 32
3.2.2 Virtual Environment Editor 32
3.2.3 Rendering 34
3.2.4 Scene Control 36
3.2.5 Agent Community 37
3.2.6 Agent Platform 38
3.3 Reverse Engineering of GeDA-3D 44
3.3.1 The Context Module 44
3.3.2 Virtual Environment Editor 45
3.3.3 Rendering 45
3.3.4 Scene Control and Environment Representation 46
3.3.5 Agent Architecture 46
3.3.6 Agent Platform 47
3.3.7 GeDA-3D Architecture 49
3.4 Reengineering of GeDA-3D 50
3.4.1 The Context Module 50
3.4.2 Virtual Environment Editor 51
3.4.3 Agent Community 51
3.4.4 Agent Platform 53
3.4.5 The Distributed Environment 53
3.5 Summary 54
Chapter 4 Distributed System Platform 55
4.1 Introduction 55
4.2 Architecture 56
4.3 Microkernel 57
4.4 Process and Thread Administration 61
4.5 Platform Services 63
4.5.1 Thread Management 65
4.5.2 Process Addressing 66
4.5.3 Message Passing Primitives 68
4.5.4 Message Storage and Delivering 69

4.5.5 Reliability 70
4.5.6 Distributed Mutual Exclusion 71
4.5.7 Group Communication 72
4.6 Agent Platform Services 74
4.6.1 Life Cycle Management 75
4.6.2 White and Yellow Page Services 76
4.6.3 Message Transport Service 78
4.7 Summary 79
Chapter 5 Adaptive Distributed Multiagent Environment 81
5.1 Introduction 81
5.2 Architecture 82
5.3 Distributed Virtual Environment 87
5.4 Dynamic Adaptation Policies 93
5.5 Summary 99
Chapter 6 Case Study 101
6.1 Introduction 101
6.2 Prey Predator Ship Battle 102
6.2.1 Illustrating the case without using dynamic adaptation policies 102
6.2.2 Metrics without using dynamic adaptation policies 111
6.2.3 Metrics applying dynamic adaptation policies 113
6.3 Prey Predator Avatar Chasing 116
6.3.1 Illustrating the case without using dynamic adaptation policies 116
6.3.2 Metrics without using dynamic adaptation policies 126
6.3.3 Metrics applying dynamic adaptation policies 128
6.4 Prey Predator Avatar Chasing using the Scenario Descriptor 131
6.5 Summary 132
Chapter 7 Conclusions and Future Directions 133
7.1 Conclusions 133
7.2 Future Directions 137
Bibliography 141

List of Figures

Figure 1-1. Distributing modules Scenario and Context 3
Figure 1-2. Environment Agent in charge of region r and its modules 4
Figure 1-3. The Environment divided into regions of volume *x* 4
Figure 2-1. A distributed system organized as middleware 8
Figure 2-2. Remote placement of work 11
Figure 2-3. Replication of an item *x* provided by a server to be read by a client 12
Figure 2-4. Alternatives to achieve code migration 13
Figure 2-5. DSM (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used. 14
Figure 2-6. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol. Extracted from [TANENBAUM] 15
Figure 2-7 Object-based distributed shared memory. 16
Figure 2-8. An agent and its environment 16
Figure 2-9. Centralized and Distributed Environments (a) centralized (b) distributed. Extracted from [ELMERHEBI] 17
Figure 2-10. RING servers (a) RING servers managing communication. (b) Cell to cell visibility. Extracted from [FUNKHOUSER] 20
Figure 2-11. The Contract net (CNET) protocol 21
Figure 2-12. The Model-View-Controller design pattern. Extracted from [ORACLE] 25
Figure 3-1. ViSCA seen as a black box 31
Figure 3-2. Initial GeDA-3D architecture 31
Figure 3-3. Scene Descriptor 33
Figure 3-4. Display of ViSCA Render 35
Figure 3-5. Display of AVE-3D Render 35
Figure 3-6. Graphical emotional state. (a) Fear (b) Happiness 36
Figure 3-7. Action-Reaction cycle in the Scene Control 37
Figure 3-8. GeDA-3D Agent Architecture 38
Figure 3-9. Virtual Scene Creator Architecture (ViSCA) 39

Figure 3-10. FIPA Agent Abstract Architecture 39
Figure 3-11. GeDA-3D Core 40
Figure 3-12. Agent Administrator 42
Figure 3-13. Agent definition view 42
Figure 3-14. Human-Agent interaction 43
Figure 3-15. Virtual Environment Editor 45
Figure 3-16. FIPA Agent Life Cycle. Extracted from [FIPA_SPEC] 48
Figure 3-17 New GeDA-3D Architecture 51
Figure 4-1. Middleware Architecture 56
Figure 4-2. Relationship between Kernel and threads 61
Figure 4-3. Platform Services 65
Figure 4-4. Bit setting to distinguish kernel packets, process IDs and group IDs...68
Figure 4-5. The Critical Region Coordinator 72
Figure 5-1. Distributing modules Scenario and Context 82
Figure 5-2. Environment Agent in charge of region *r* and its modules 83
Figure 5-3. The Environment divided into 27 cubes 83
Figure 5-4. The Scenario Module 83
Figure 5-5. The Generic Environment 84
Figure 5-6. The Generic Environment Object 85
Figure 5-7 The Cube 86
Figure 5-8. The Environment Agent 86
Figure 5-9. The Environment divided into regions of volume *x* 88
Figure 5-10. The Distribution Map 89
Figure 5-11. One EA administering the entire environment 90
Figure 5-12. Dichotomy process on EA_1 creates EA_2 90
Figure 5-13. Dichotomy process on EA_2 creates EA_3 90
Figure 5-14. Dichotomy process on EA_2 creates EA_4 91
Figure 5-15. Dichotomy process on EA_1 creates EA_5 91
Figure 5-16. Fusion process of EA_1 with EA_4 and fusion of EA_2 with EA_5 91
Figure 5-17. Three intentions with an increase of time of response. 92
Figure 5-18. Activity diagram to determine the best partition during dichotomy process (part 1). 96
Figure 5-19. Activity diagram to determine the best partition during dichotomy process (part 2). 97
Figure 6-1. Start of the GeDA-3D Distributed System Platform 102
Figure 6-2. The scene and scenario descriptions for the first case study. 103
Figure 6-3. The Environment Agent at the beginning of the first case study. 103
Figure 6-4. The Scenario's state: at the beginning of the first case study. 104
Figure 6-5. Core in the first case study 104

Figure 6-6. The Agent Administrator of the first case study. 105
Figure 6-7. Ship1 at the beginning knows 5 objects of the environment. 105
Figure 6-8. Ship2 at the beginning knows 5 objects of the environment. 105
Figure 6-9. The scenario after 77 seconds since the beginning. 106
Figure 6-10. The scenario after 274 seconds since the beginning. 106
Figure 6-11. Ship1 knows 12 objects of the environment. 107
Figure 6-12. Ship2 knows 7 objects of the environment. 107
Figure 6-13. The scenario after 437 seconds since the beginning. 108
Figure 6-14. The scenario after 478 seconds since the beginning. 108
Figure 6-15. The Environment Agent after 478 seconds since the beginning. 109
Figure 6-16. Ship1 reaching the goal attack. 109
Figure 6-17. The scenario at the end of the simulation of the first case study. 110
Figure 6-18. Ship1 after the end of the simulation of the first case study. 110
Figure 6-19.The Environment Agent after the end of the simulation of the first case study. 111
Figure 6-20. Objects Administered using one EA in the first case study. 111
Figure 6-21. Average Region Load using one EA in the first case study. 112
Figure 6-22. Time of Response using one EA in the first case study. 112
Figure 6-23 Percentage of Time of Service using one EA in the first case study. 113
Figure 6-24. Amount of agents using up to two EAs in the first case study. 113
Figure 6-25. Objects Administered using up to two EAs in the first case study. .. 114
Figure 6-26. Average Region Load using up to two EAs in the first case study. .. 114
Figure 6-27. Time of Response using up to two EAs in the first case study. 115
Figure 6-28 Percentage of Time of Service using up to two EAs in the first case study. 115
Figure 6-29. The prototype of the Scene Descriptor version 2 for the second case study. 116
Figure 6-30. The Environment Agent at the beginning of the first case study. 117
Figure 6-31. The Scenario´s state: at the beginning of the second case study. ... 117
Figure 6-32. Agent luis receives its goal specification. 118
Figure 6-33. Agent albert receives its goal specification. 118
Figure 6-34. The scenario after 40 seconds since the beginning. 119
Figure 6-35. The Environment Agent after 40 seconds since the beginning. 119
Figure 6-36. The scenario after 75 seconds since the beginning. 120
Figure 6-37. The Environment Agent after 75 seconds since the beginning. 120
Figure 6-38. Agent luis knows its position x=140, y=0, z=125. 121
Figure 6-39. Agent albert knows its position x=-110, y=0, z=122. 121
Figure 6-40. The scenario after 103 seconds since the beginning. 122
Figure 6-41. Agent albert reached its goal. 122

Figure 6-42. The scenario after 165 seconds since the beginning. 123
Figure 6-43. The scenario after 188 seconds since the beginning. 123
Figure 6-44. The scenario after 220 seconds since the beginning. 124
Figure 6-45. The scenario after 250 seconds since the beginning. 124
Figure 6-46. The Environment Agent after 250 seconds since the beginning. 125
Figure 6-47. Agent luis reached its goal. 125
Figure 6-48.The Environment Agent after the end of the simulation of the second case study. 126
Figure 6-49. Objects Administered using one EA in the 2nd case study. 126
Figure 6-50. Average Region Load using one EA in the 2nd case study 127
Figure 6-51. Time of Response using one EA in the 2nd case study. 127
Figure 6-52 Percentage of Time of Service using one EA in the 2nd case study. 128
Figure 6-53. Amount of agents using up to two EAs in the 2nd case study. 128
Figure 6-54. Objects Administered using up to two EAs in the 2nd case study. .. 129
Figure 6-55. Average Region Load using up to two EAs in the 2nd case study... 129
Figure 6-56. Time of Response using up to two EAs in the 2nd case study. 130
Figure 6-57 Percentage of Time of Service using up to two EAs in the 2nd case study. 130
Figure 6-58. The Scenario Descriptor for the third case study. 131
Figure 6-59. The Scenario Descriptor after compiling the described scenario. 131
Figure 6-60. The scenario at the beginning of the third case study. 132

Glossary of Terms and Acronyms

AAG	Acknowledge Action for Group
ACK	Acknowledgement
ACL	Agent Communication Language
ADT	Abstract Data Type
AGA	Agree Group Address
AMS	Agent Management System
ASN	Agent Service Name
AU	Address Unknown
AUN	Agent Unique Name
AYA	Are You Alive?
CNET	Contract Net
DF	Directory Facilitator
DM	DistributionMap
DSM	Distributed Shared Memory
EA	Environment Agent
EO	EnvironmentObject
FGA	Found Group Address
FIPA	Foundation for Intelligent Physical Agents
FSA	Found Service Address
GeDA-3D	Generic Distributed Architecture for 3D Applications
GEO	GenericEnvironmentObject
GID	Group Identifier
IAA	I Am Alive
IP	Internet Protocol
IPR	Is Platform Running
JVM	Java Virtual Machine
KTK	Kernel To Kernel
LCL	Local Class Loaded
LGA	Lookup Group Address
LIME	Linda in a Mobile Environment
LLC	Load Local Class
LSA	Lookup Service Address
MC	Module Context
MS	Module Scenario
MVC	Model-View-Controller

NKL	Notify Kernel Load
OKB	Operating Kernel Bye
OKP	Operating Kernel Presentation
PGA	Propose Group Address
PID	Process Identifier
PIR	Platform Is Running
PSN	Process Service Name
PUN	Process Unique Name
RAG	Remote Action for Group
RGA	Reject Group Address
RPC	Remote Procedure Call
SDE	Scenario Descriptor Emulator
SID	Subsystem Identifier
SPID	Subsystem Process Identifier
SUN	Subsystem Name
TA	Try Again
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UGA	Unknown Group Address
VEE	Virtual Environment Editor
ViSCA	Virtual Scene Creator Architecture
VR	Virtual Reality
XML	Extensible Markup Language

Chapter 1

Introduction

This chapter presents a summary about what this dissertation contributes with, first describing the problem to solve and our motivation, followed by the goals to accomplish in order to make a proposal for an Adaptive Distributed Multiagent Environment; finally it is shown the structure of this thesis.

1.1 Problem Description

The problem we try to solve in this dissertation is double. The first deals with **reverse engineering our GeDA·3D architecture** to obtain documentation and software components and interfaces; the second is **alleviating a problem of bottleneck** our GeDA.3D architecture has.

Virtual Reality is represented by different interacting entities allocated into an environment, and such entities are designed to carry out some task.

In a Multiagent System, the Environment is the entity that contains information of the environment's state and may also contain rules about how to modify it

Context, in our approach, is the entity that contains information about the Environment where an agent community activity is developed. For example, the context in the real world contains physical laws that rule our world, semantic concepts of words, relationships between existent entities in the world, etc.

Scenario is the environment where a scene is carried out. The Environment is represented by a Scenario and a Context together.

During scene development, agents evolve according to their goals and perceptions about the environment, objects and other agents around them. This means that each agent consults the environment to perceive from it, and also to notify it any intentions to carry out modifications to its state, as well as to validate in the context the rules for permitted changes.

Administration of a dynamic 3D environment is difficult because the continuous communication of all agents to the environment; such communication may turn the environment into a bottleneck for the whole system, because the context inside the environment may take considerable computer resources for validation purposes, causing the agents do not perceive changes in the environment effectively. To alleviate this bottleneck problem, we propose to distribute the scenario and the context adaptively into a set of agents.

GeDA-3D (Generic Distributed Architecture for 3D Applications) [RAMOS] is a platform useful to manage distributed applications; it provides several services to manage communication among agents and the virtual environment evolution according to virtual objects behavior and laws ruling the interactions; it also provides mechanisms to translate the specification of the environment into a set of primitive commands used for the rendering. This platform is the core in charge of handling the scene, since its creation until its final rendering.

Around of GeDA-3D project architecture, there have been many developments, which unfortunately have not all been integrated to the platform due to the lack of well implemented and independent software components and interfaces, applying software engineering methods during the construction of the software. Those developments have been tested in an isolated manner, being the most recently achievement the integration of three of these component, but leaving the components with a considerable level of coupling, a minimum level of cohesion and far from meeting the Model-View-Controller design pattern; the latter could allow future improvements to the platform, replacement of specific modules and the integration of new components.

Analyzing the subject system in order to solve the problem of the bottleneck we also found that the kernel was implemented forming a mesh topology between applications, with a connection oriented protocol from one application to each one of the others, which causes problems to the scalability of the system and to the agents mobility through the network; therefore it was mandatory the update of some components, and a reengineering (reverse engineering and restructuring) of the currently implemented software components including modifications to the GeDA-3D architecture.

1.2 Goals

The first goal of this work is transforming the current platform of GeDA-3D into a flexible and scalable platform where the previously developed applications can still be runnable, preserving their functional requirements and expected results of execution. The new platform design must provide all general services of a middleware, and also satisfy the requirements to achieve distribution of a centralized component, such as the environment of a multiagent system.

The GeDA-3D's environment currently implants two of its modules, the scenario (MS) and the context (MC) in a centralized manner.

The final goal is to distribute both modules through available processors (or machines) over the network, so an entity (an agent) only represents one part of the environment as illustrated in Figure 1-1.

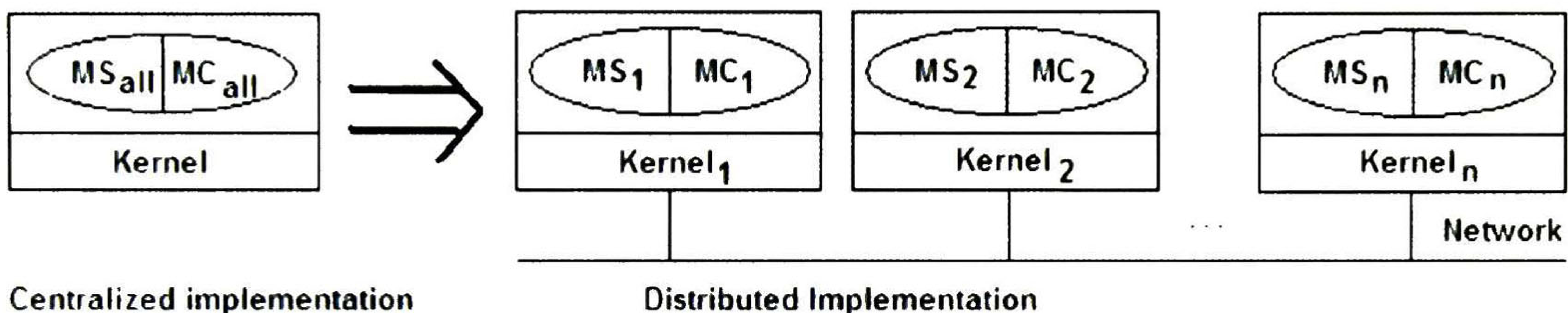


Figure 1-1. Distributing modules Scenario and Context

The distribution proposed must assign (computer) resources in function of the complexity of the scenario at runtime. As the environment changes as time goes on, such resource assignment must be done dynamically.

1.3 Proposal

In this thesis is proposed a solution to solve the problem of the bottleneck, to which the entity that represents the environment tends to turn into.

We propose an administration of the environment based on special agents denominated Environment Agents. An Environment Agent is in charge of:

- receiving any agent intentions to modify the environment´s state,
- validating applicable changes into the context,
- knowing each entity relationship with others and,
- propagating all permitted changes only to agents interested in such changes

All these activities increase workload of the Environment Agent.

In order to distribute load, an Environment Agent may decide to apply a dichotomy to it, by means of cloning. It divides its information and workload with its

clone agent. Vice versa, if it finds itself almost or already idle, it may decide to fuse with another Environment Agent. This behavior is achieved in function of the complexity of the 3D environment.

The design principle of GeDA-3D environment distribution is based on Environment Agents representing both modules Scenario (*MS*) and Context (*MC*). An Environment Agent (EA) is in charge of administering a region *r* consisting of a MS partition and a MC cache as illustrated in Figure 1-2.

Figure 1-2. Environment Agent in charge of region r and its modules

The proposed solution in this work consists in distributing the environment adaptively on a set of Environment Agents as illustrated in Figure 1-3.

Each Environment Agent represents a volume *x* of the environment and, all of them are perceived as a single entity by other agents.

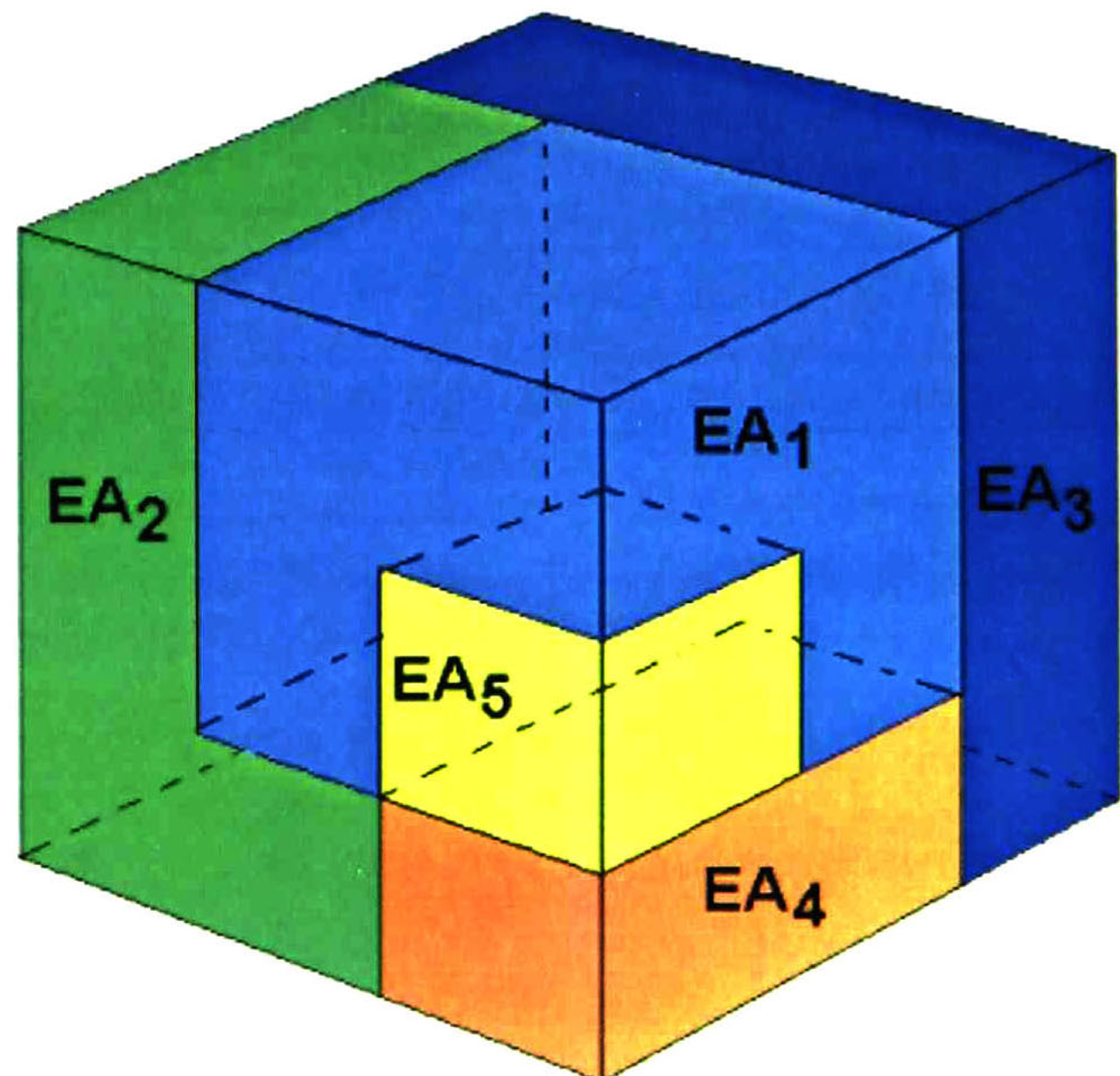


Figure 1-3. The Environment divided into regions of volume *x*

We say that distribution is adaptive, because the number of agents that represent the environment is calculated in function of the complexity of the environment for each time *t*. If the environment is complex, then we will have that the number of Environment Agents is big, otherwise is small.

More precisely, distribution is done in function of the complexity of each part (a region *r*) of the environment. We will have more Environment Agents (resolving the bottleneck) in sections of the environment that contain more complexity and, a smaller number in sections with low complexity.

When an Environment Agent decides to apply a dichotomy (cloning itself for another instance), this decision is based on an important increase of complexity it manages. Our solution resolves the problem for deciding where to generate a new Environment Agent and what information is provided to it; in a similar way, if

complexity diminishes, two or more Environment Agents will agree to fuse their information.

The proposed solution is integrated as two modules of the GeDA-3D project architecture. These modules are constituted by a middleware, with support functions for threads and reliable communication and, a set of generic Environment classes written in Java language. The decision of using Java besides being a multiplatform language is because the language lets inheritable specifications and, therefore, reusable for any kind of specific environment that is desired to put in scene on the GeDA-3D platform.

1.4 Structure of this Thesis

This dissertation is organized as follows:

Chapter 2 State of the Art exhibits all fundamentals useful for the realization of this work, such as distributed system platforms, multiagent system environments and software engineering.

Chapter 3 Software Engineering of GeDA-3D analyzes through re-documentation and reverse engineering, current components developed for the GeDA-3D project; this analysis is needed to execute a reengineering of these components to integrate them with the new kernel that supports the requirements for the distributed environment.

Chapter 4 Distributed System Platform specifies the architecture and system requirements for our distributed system platform, in order to support any distributed application, specially a multiagent system that needs to distribute a centralized component like the Environment.

Chapter 5 Adaptive Distributed Multiagent Environment describes our proposal, applying techniques of distributed shared memory and load sharing, to distribute and locate information while executing dynamic adaptation policies at runtime, in order to control workload for the *n* agents that will represent the environment.

Chapter 6 Case Study presents graphical results and metrics of the behavior of the implemented environment, by means of the execution of three cases and, using all developed components for GeDA-3D integrated through the platform.

Chapter 7 Conclusions and Future Directions summarizes our thoughts about the work done, goals achieved, improvements to the GeDA-3D architecture, contributions to the state of the art of distributed multiagent systems and presents ideas for future work to extend ours.

Chapter 2

Review of Related Works

In this chapter we describe all fundamentals useful for the realization of this work, such as distributed system platforms, multiagent system environments and software engineering which are strongly related to the development of the work presented in this thesis.

2.1 Introduction

Virtual reality is the representation of an actual or fictitious evolving environment using digital devices. Unanimated objects like tables, walls, etc. constitute the virtual world or environment, and one or many virtual entities or avatars evolve in order to reach one or different goals. To implement the control of virtual creatures, or avatars, we use the agent's paradigm. Agents perceive from the environment, and based on their goals, make a decision about how to evolve in the environment. Any intention is validated in the environment, and changes are propagated only to agents that are interested in such changes. The necessity for distributing the environment is because some agent can need high quantities of processing, besides, applications can be distributed by nature.

2.2 Distributed Systems

A distributed system is a collection of independent computers that appears to its users as a single coherent system [TANENBAUM_STEEN]. The definition considers two aspects: autonomous components and that its users (people or programs) think they are dealing with a single system. Important characteristics of a distributed system are:

1) Users do not realize how the system deals with differences between computers, how they communicate and how they organize
2) Users and applications can interact in a consistent and uniform way.

2.2.1 Middleware

In a distributed system, a middleware may be necessary to make easier for a developer to implement a solution. In fact, the middleware helps to hide actual locations, all communication details, and just providing easy primitives to use; middleware also provides mechanisms to locate services and manages messages for reliable communication.

A middleware is an application running in the application layer [TANENBAUM_STEEN]. Middleware contains many general-purpose protocols, in order to offer a single system view, supporting a programming abstraction; it masks heterogeneous computers, operating systems, networks and programming languages (e.g. CORBA) [COULOURIS].

Distributed systems are often organized by means of a Middleware that is placed between the application layer and the operating system; see Figure 2-1.

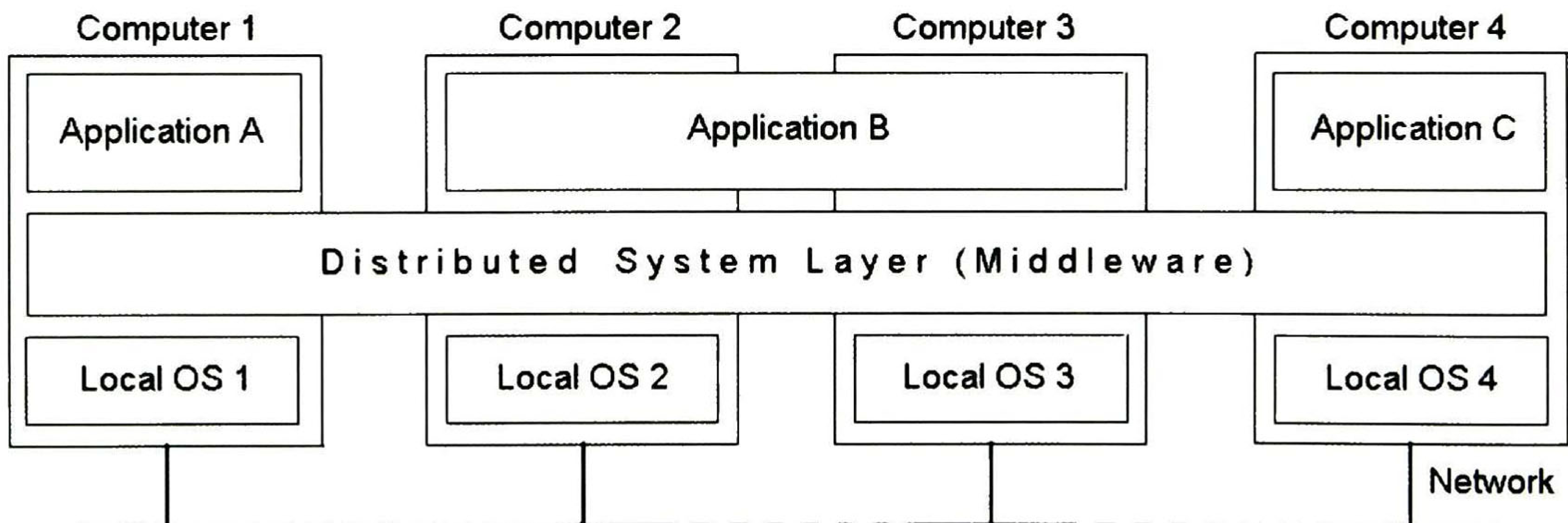


Figure 2-1. A distributed system organized as middleware

Middleware replaces the corresponding session and presentation layers of the OSI model [TANENBAUM_STEEN], it is in charge of fulfill solutions to key design issues of distributed systems [TANENBAUM] such as transparency of: access, location, migration, relocation, replication, concurrency, parallelism, failure, flexibility, reliability, performance and scalability.

Middleware helps to develop under client-server model, in order to avoid the considerable overhead of the connection-oriented protocols, such as OSI or TCP/IP. Client-server model is based on a simple connectionless request/reply protocol [TANENBAUM] (the OSI session layer).

Four design issues have to be considered in client-server model [TANENBAUM].

First, the main approaches of addressing, to lookup a server via software are 1) sparse process addresses (using broadcasting of a lookup packet) and 2) use of a name server.

Second, the message-passing primitives (send and receive) may be chosen to be blocking or nonblocking, and over an operating system that supports multithreaded processes, the first choice is the best because it is simple to understand, simple to implement and it does not require kernel buffers to manage.

Third, unbuffered primitives require that destination server is ready waiting for a message, reason why it is preferable a buffered primitive using a data structure called a *mailbox*, in order to avoid messages being discarded, at least as long as it is not filled up, in such case maybe we could require keeping incoming messages, for a little while until an appropriate receive is done shortly.

Fourth, reliable primitives using special types of packets to implement client-server protocols, such as an acknowledgement (ACK) package, to notify a sender for a received message; other packages like AYA (are you alive?), IAA (I am alive), TA (try again) and AU (address unknown) serve to make the best effort for a message delivering to its destination. Reliability may be provided in two variants: request-ACK-reply-ACK or request-reply-ACK; the latter considers reply as and ACK saving one packet; an acknowledgement to a request lets that a sender is informed of message delivering faster.

In order to obtain that a middleware supports heterogeneity, it includes a presentation layer that is in charge of the extern data representation, packing parameters into a message (called parameter marshalling), to carry out an RPC (Remote Procedure Call) [COULOURIS]

2.2.2 Load Sharing and Load Balancing

For distributed environment policies it is important the following analysis about how and when disseminating information to interested nodes. With decision making is analyzed if it is appropriate to initiate migration of work from the overloaded node or from the underloaded node.

In order to improve performance of a distributed system, load sharing policies attempt to assure that no node is idle while other software components wait for service [EAGER], on the other hand load balancing algorithms strive to equalize the workload among nodes.

Krueger and Livny [KRUEGER] show that load balancing strategies put much higher resource requirements on the system than load sharing strategies, and that these resource requirements may outweigh their potential benefits

Analysis of Adaptive Load Sharing Algorithms, presented by [KREMIEN], shows a method for qualitative and quantitative analysis of load sharing algorithms that comprises the following characteristics:

a) Considering information dissemination and allocation decision making
b) Nodes must be capable of making local decisions
c) Remote execution should be bounded and restricted to a small proportion of system activity

Load sharing is the problem of allocation: of mapping and remapping the logical system (running applications), to the physical system (processing nodes interconnected by a communication network). A flexible load-sharing algorithm is required to be general, adaptable, stable, scalable, transparent to the application, fault tolerant, and induce minimum overhead on the system.

<u>Qualitative analysis</u>

In a qualitative analysis, adaptive algorithms for load sharing comprise two main activities: Information dissemination and decision-making (control).

Information dissemination queue length is the state metric used by the algorithms studied [FERRARI, ZHOU87]. We have to decide whether to hold state information regarding all nodes in the system, or only a subset; for scalability purposes, subsets are used and criteria specification is needed; a node may choose to request information, disseminate information or use a predictive analysis technique (objective: minimize overhead). With state dissemination the node may have the information available when it is needed; algorithms using periodic (choose frequency) and event-driven (significant change) information dissemination policies, provide comparable performance. The use without any policy of broadcast [LIVNY, ZHOU88] or multicast [THEIMER] results in intolerable overheads.

In decision-making, the remote placement or migration can be initiated by the source of work (overloaded node), or the server of work (underloaded node). We have two options: a source initiated algorithm (single request, request and reply) and a server initiated algorithm (request and reply). Source-initiated algorithms place much of the overhead on the overloaded nodes, whereas server-initiated algorithms on the underloaded nodes are more stable, controlling the amount of work requested; see Figure 2-2.

Negotiation enables poor decisions to be tolerated by inhibiting their consequence. A bad decision can be reversed; but when it is not possible to attach the work to the request, the negotiation is compared with a single request protocol, and the number of involved phases is larger requiring more delay.

A combination of source-initiated and server-initiated policies can give the best results. About when decision-making should be activated we have two options: periodic and event based. With periodic decision-making it is hard to select a sensible time period, since it is very dependent on the loads on the system. In [KREMIEN] is advocated the use of a combination of both approaches, with events as the basic method, and a lower bound periodicity selected to prevent overuse and overload. A detailed quantitative analysis can be found at [KREMIEN].

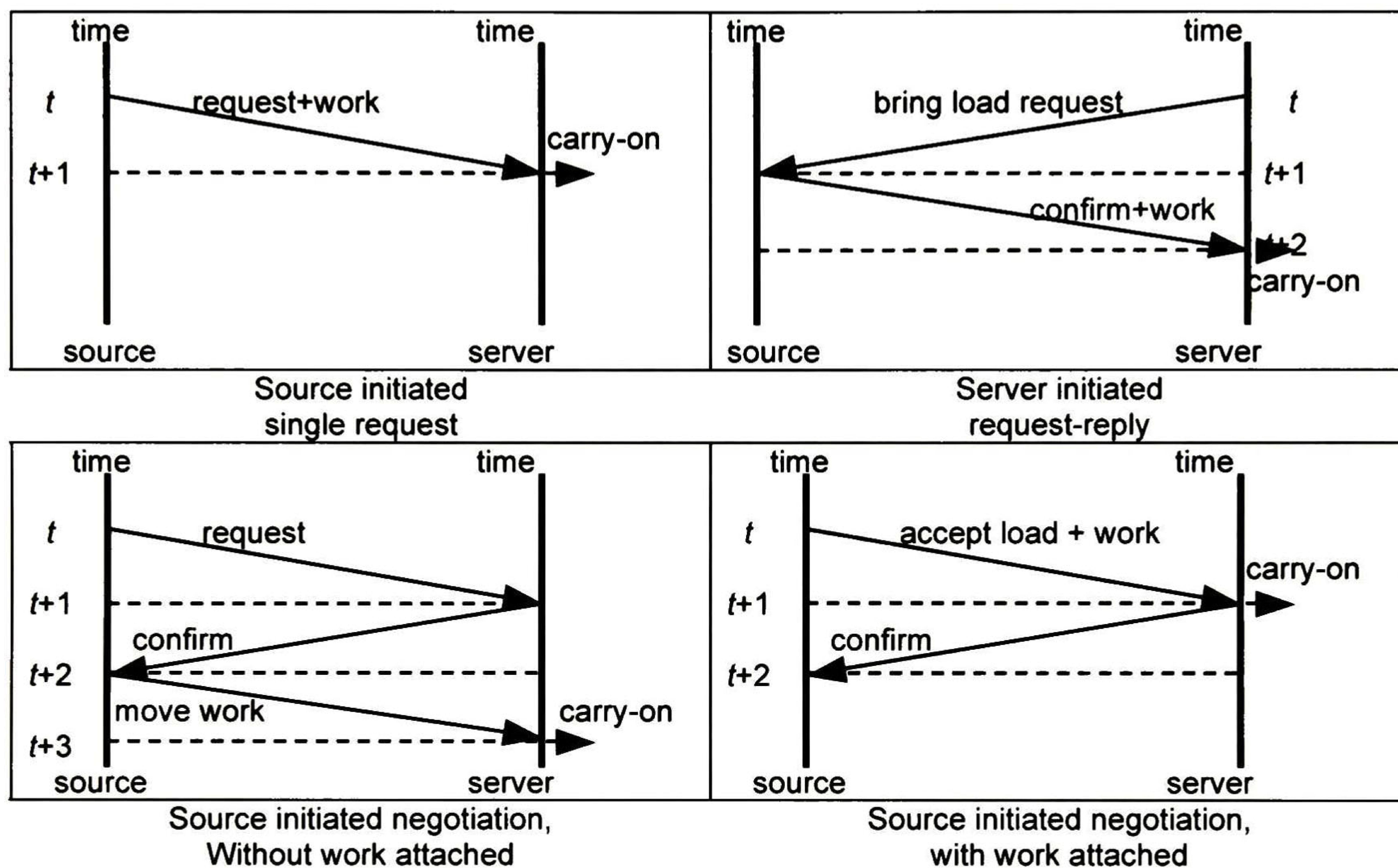


Figure 2-2. Remote placement of work

2.2.3 Replication and Mobility

Our distributed environment proposal uses copies of information provided by a server; such server and environment agents are assumed as mobile for load balancing policies; information stored in the server may change, so the following paragraphs show different options we have.

Replication is the key approach to provide high availability and fault tolerance in distributed systems, given an increasing interest to mobile computation [COULOURIS].

Considering a hierarchical network with L levels of location servers, and a single server and a single client who move entirely within their location servers, let s be a server and c be a client, and let x be an item which is written by a server s and read by the client c, an illustration of this situation is shown in Figure 2-3.

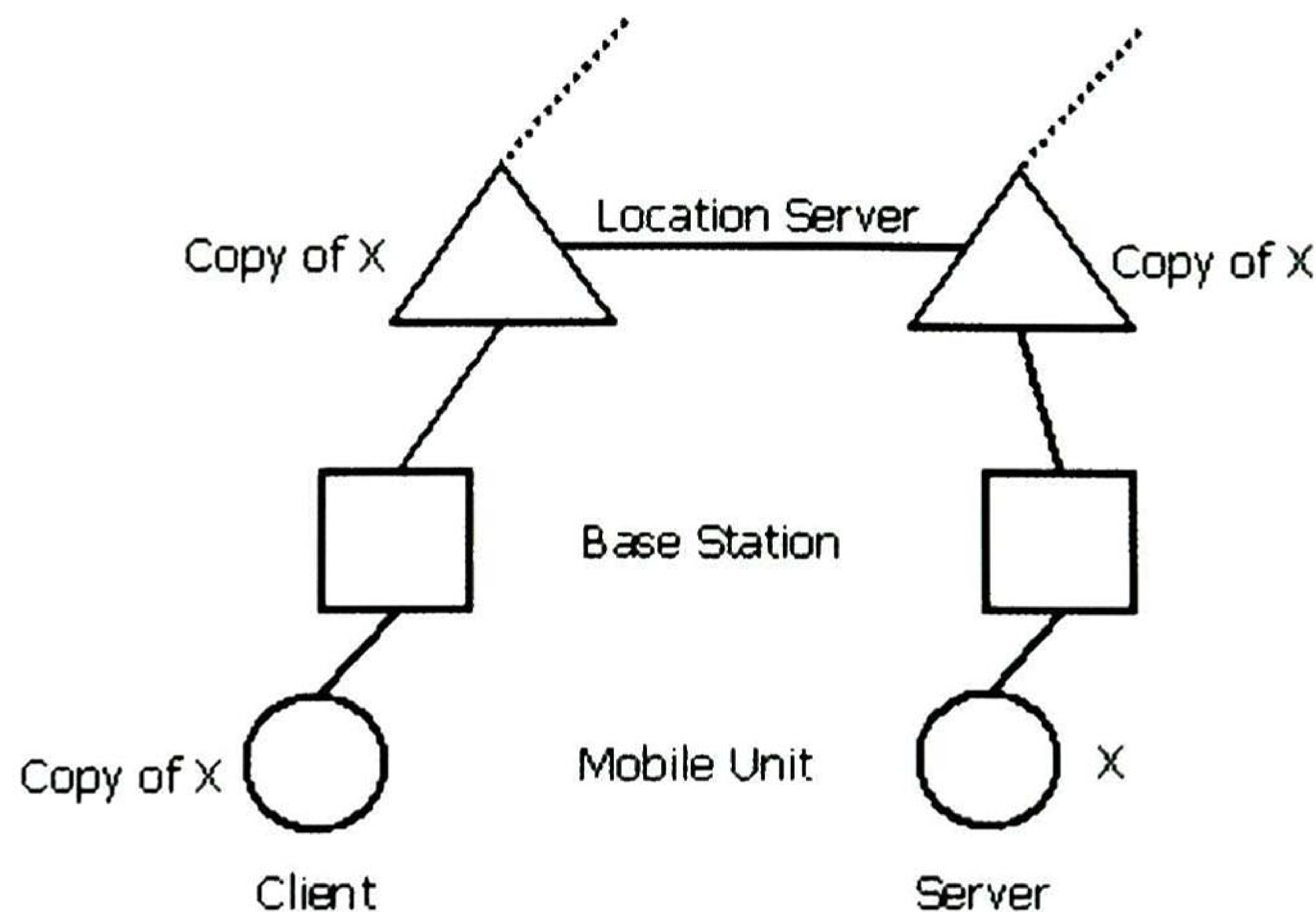


Figure 2-3. Replication of an item *x* provided by a server to be read by a client

According to [BADRINATH] we may have 6 kinds of replication schemes. The first three schemes do not use a caching technique. A copy of the server data is contained in either of the location servers or on the mobile client. In the latter three schemes, a copy of the data is cached at various places, and update messages to the copy are deferred until an access of a previously invalidated data occurs. The following are the choices:

1. The server replicates the copy of the data at the mobile client. On each write, the server needs to write to the copy on the mobile client. Writing requires locating the mobile client. Reading is from a local copy on the mobile client.
2. The replicated copy resides at the location server of the client. Thus, the client reads from its own location server. Here, reads and writes are on static copies. However, the copy is closer to the reader than the writer.
3. The server *S* has a copy of the data at its home location server L_S. The client reads from L_S. Thus, reading and writing is on static remote copies.
4. The server *S* has a cached copy at its home location server. Location server copy is invalidated upon the first write, since the last read request from the client. Reading an invalid cache will require locating the mobile server. However, if the cache is valid, then the read takes place from the copy at the location server L_S.
5. A cache of the server's copy exists at the client's location server. Location server copy is invalidated upon the first write, since the last read request from the client. Reading an invalid cache will require locating the mobile server. However, if the cache is valid, then the read takes place from the copy at the location server L_C.
6. The cache is maintained at the mobile client. If there was a read since the last update, the server upon each write sends invalidation message to the mobile client. If client wants to read and the cache is invalidated, then the mobile server is contacted to complete the read.

2.2.4 Code Migration

Given the fact that it is necessary to consider mobility in a distributed system, code migration has to be applied in some form in order to move running servers or clients from one location to another and, continue their execution at their target machine transparently.

Code migration, in the broadest sense, deals with moving programs between machines in order to be executed at the target. Process consists of three segments: the code segment (instructions) the resource segment (references to external resources) and the execution segment (state of the process) [TANENBAUM_STEEN].

The minimum code migration is providing *weak mobility*; this means only transferring the code segment and the program is started from one of several predefined starting positions. The benefit is simplicity, only requiring making the code portable. In *strong mobility* the execution segment is transferred as well, stopping process executions and resuming at the target; it is more general than weak mobility but harder to implement; see Figure 2-4.

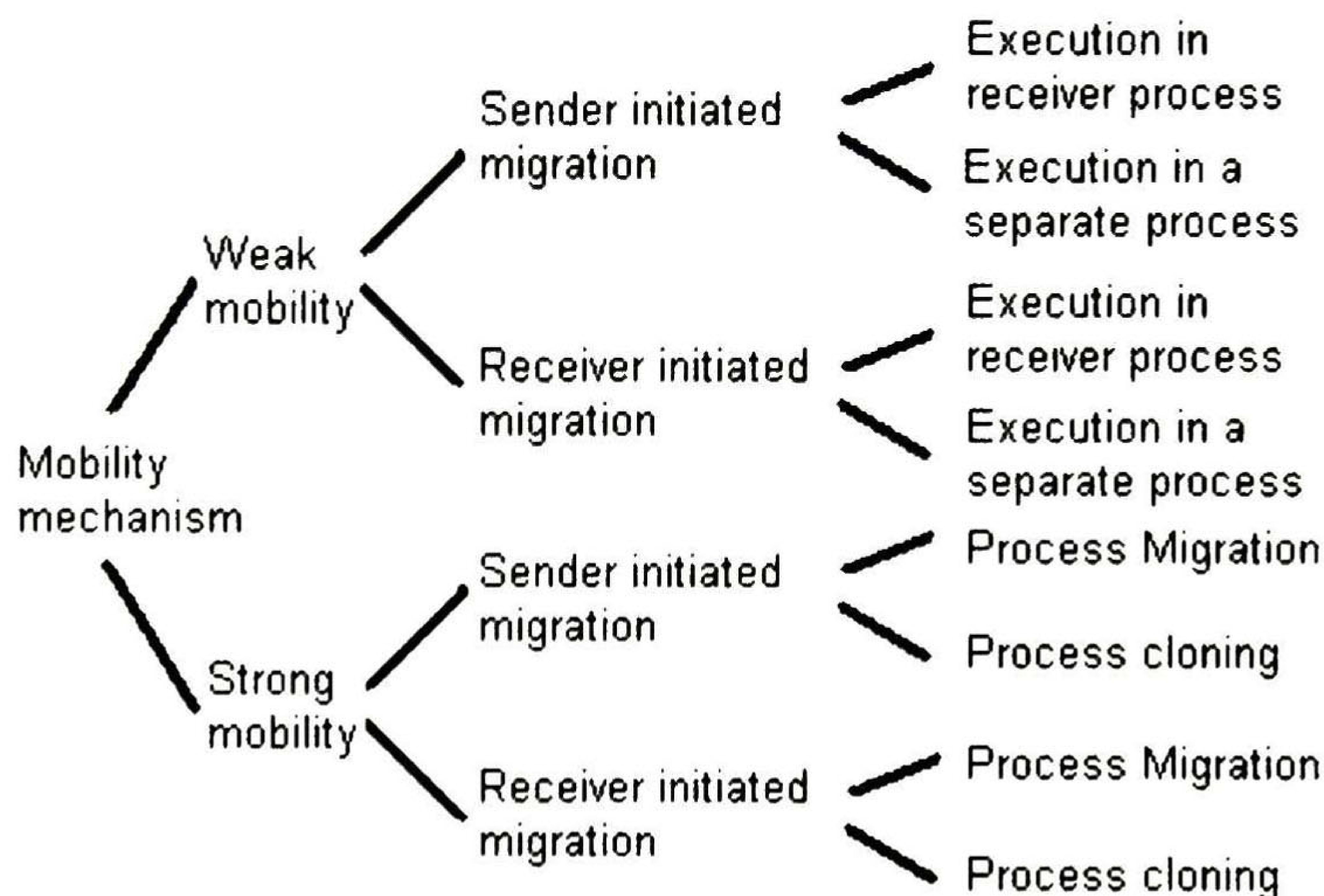


Figure 2-4. Alternatives to achieve code migration

Sender initiated migration is initiated at the machine where the code currently resides, or is being executed, for example, when uploading programs to a computer server or when sending a search program to a database server to perform queries at that server. Receiver initiated migration is when the initiative is taken by target machine, for example, java applets.

2.2.5 Distributed Shared Memory

Shared Memory is necessary for some algorithms to work. Our collection of environment agents manages the representation of the environment as a shared memory space. Without matter if required information is stored or not, in the agent´s local memory, all agents should be able to obtain information required whoever is the owner of such information.

The scheme under the name distributed shared memory (DSM) [TANENBAUM] consists in having a collection or workstations connected by a LAN and sharing a single paged, virtual address space. In the simplest variant, each page is present on exactly one machine. A reference to a local page is done in hardware. An attempt to reference a page on a different machine causes a hardware page fault, which traps to the operating system. The operating system then sends a message to the remote machine, which finds the needed page and sends it to the requesting processor. The faulting instruction is then restarted; see Figure 2-5.

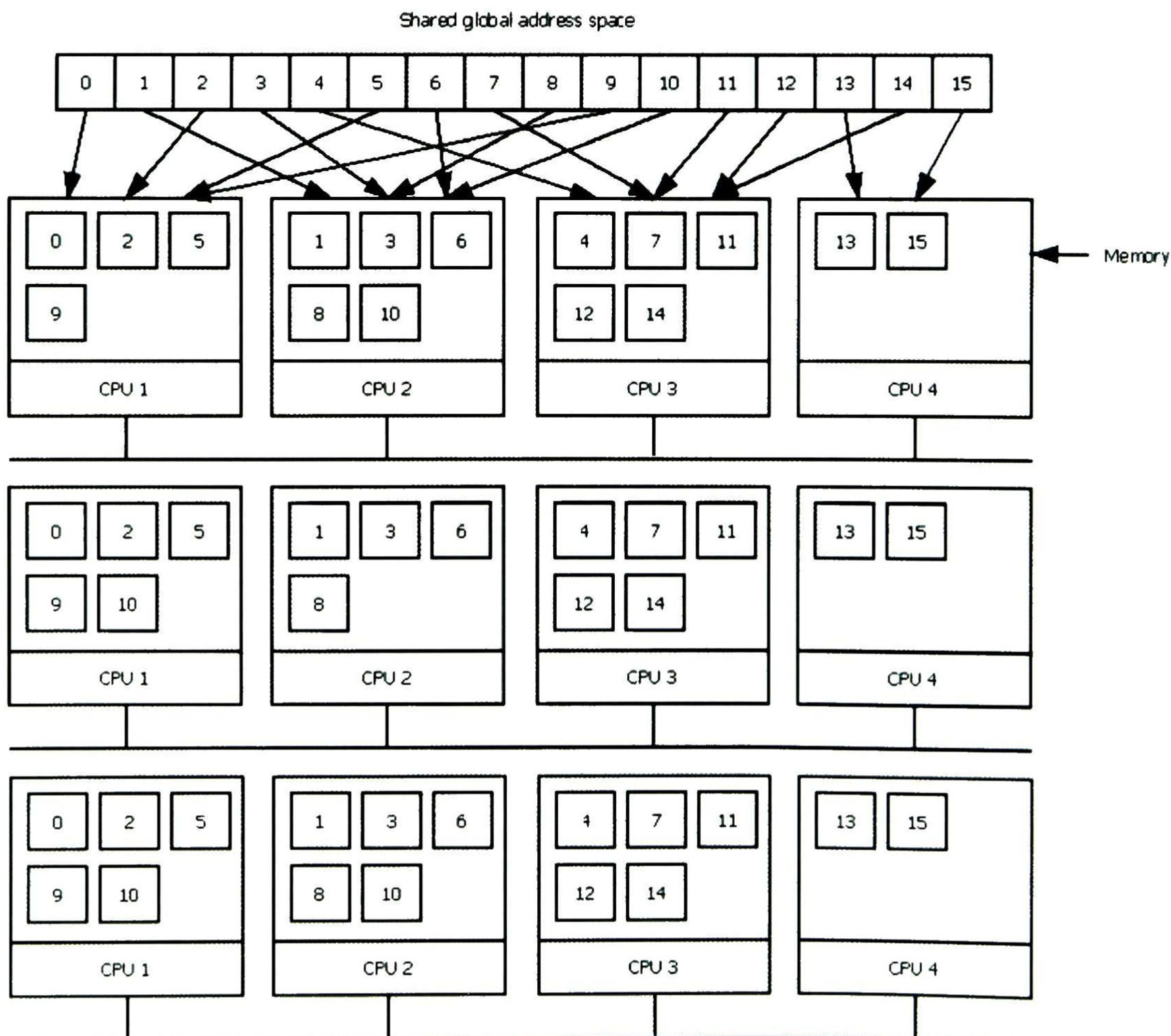


Figure 2-5. DSM (a) Chunks of address space distributed among four machines. (b) Situation after CPU 1 references chunk 10. (c) Situation if chunk 10 is read only and replication is used.

One improvement to the DSM basic system is to replicate chunks that are read-only, for example, program text, read only constants and other read-only data structures; such chunks may be replicated in several machines. If not only read-only chunks are replicated, but all chunks, when a chunk is attempted to be modified, similarly to cache consistency protocols, we need to invalidate copies in the other machines and then modify the local copy.

Each page in DSM must have an owner. To locate the owner of a page one process is designated as the page manager, which keeps track of who owns each page. When a process, *P*, wants to read a page it does not have, or wants to write a page it does not own, it sends a message to the page manager telling which operation it wants to perform and which page. The manager then sends back a message telling who the owner is. *P* now contacts the owner to get the page and/or the ownership, as required. Four messages are needed for this protocol, as illustrated in Figure 2-6(a). An optimization of this protocol is shown in Figure 2-6(b). Here the page manager forwards the request directly to the owner, which then replies directly back to *P*, saving one message.

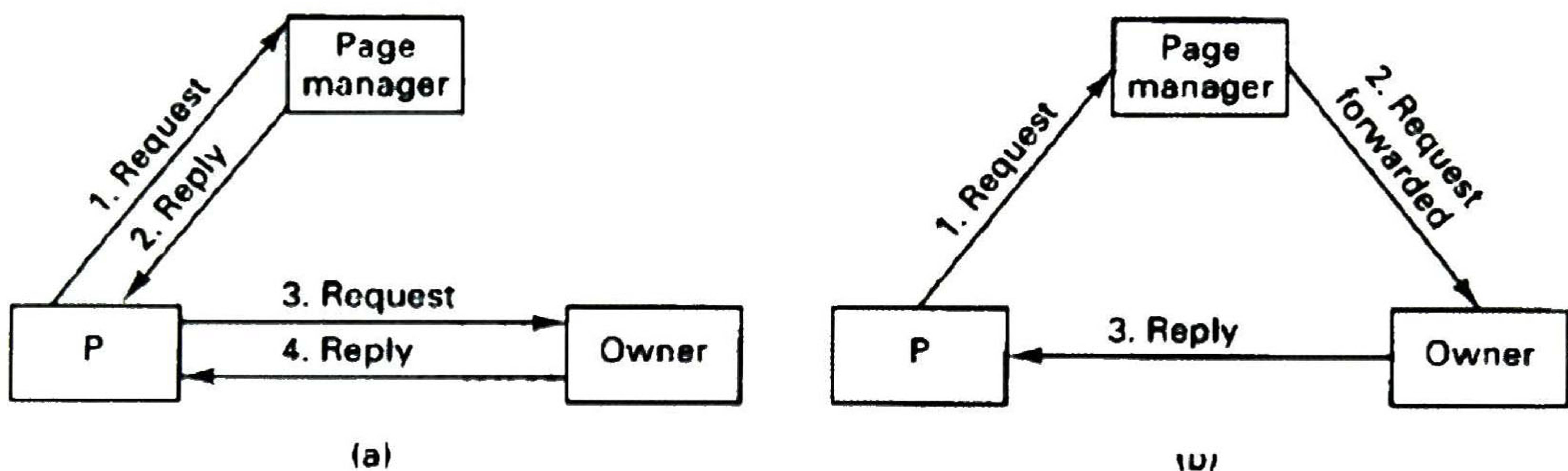


Figure 2-6. Ownership location using a central manager. (a) Four-message protocol. (b) Three-message protocol.

The problems with DSM are choosing an appropriate page size (granularity) and that for many applications it exhibits poor performance, as pages are hurled back and forth across the network, for example when a page contains two global variables, and each one is continuously used by processors in two different machines; an intelligent compiler may do some work to reduce the problem, but not so much in case where such variables are within an array.

Another method is not to share the entire address space, only a selected portion of it, namely just those variables or data structures that need to be used by more than one process. In this model, each machine has access to a collection of shared variables, and in most cases, considerable information about the shared data is available, such as their types. Besides, variables may be replicated and the problem consists in maintaining consistency of many copies. Reads can be done locally without any network traffic, and writes can be done using a multicopy update protocol. Such protocols are widely used in distributed data base systems.

Going still further, instead of just sharing variables we could share encapsulated data types, often called objects. Each object has not only some data, but also procedures, called methods, that act on the data. Programs may only manipulate an object´s data by invoking its methods. Direct access to the data is not permitted. By restricting access in this way, various new optimizations become possible. For example, it may be possible to relax the memory consistency protocol without the programmer even knowing it, because processes communicate by invoking methods on shared objects; see Figure 2-7.

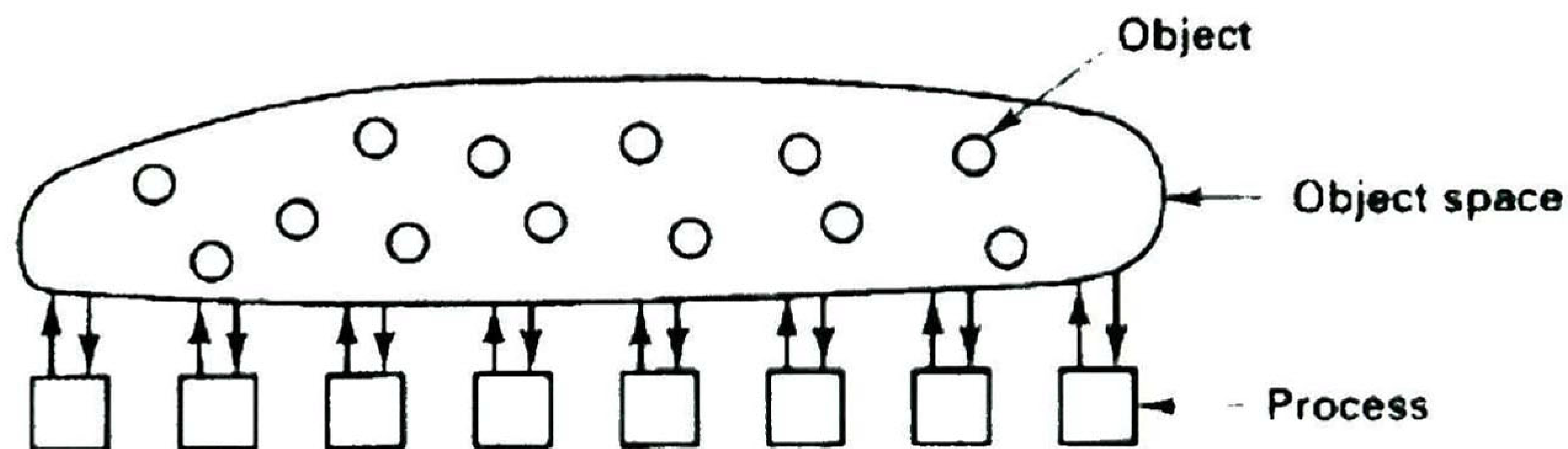


Figure 2-7. Object-based distributed shared memory.

2.3 Multiagent Systems

Our proposal consists in providing the environment representation for a multiagent system. The means and situations where agents access to the environment are necessary to be considered for the distributed environment proposal.

Multiagent systems [WOOLDRIDGE] are systems composed of multiple interacting elements, known as agents. An agent is a computer system situated in some environment, and has two important capabilities:

1) Autonomous action, to decide for itself what it needs to do in order to satisfy its design objectives; all of them on behalf of its user or owner;
2) Interacting with other agents, not simply by exchanging data, but by engaging in analogues on the kind of social activity like: cooperation, coordination, negotiation and the like.

An agent takes sensory input from the environment and produces output actions that affect it. This interaction is usually an ongoing non-terminating one, as illustrated in Figure 2-8.

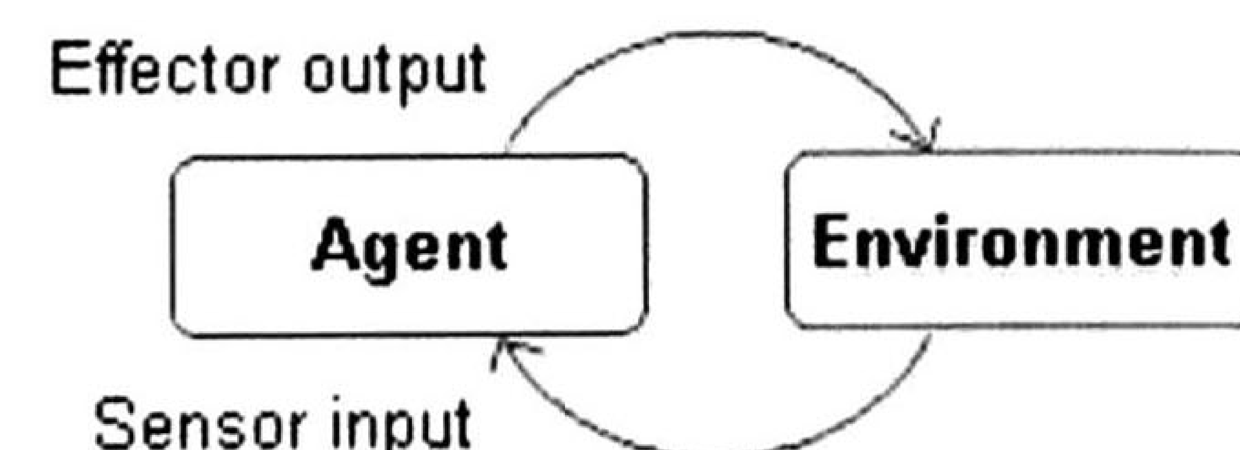


Figure 2-8. An agent and its environment

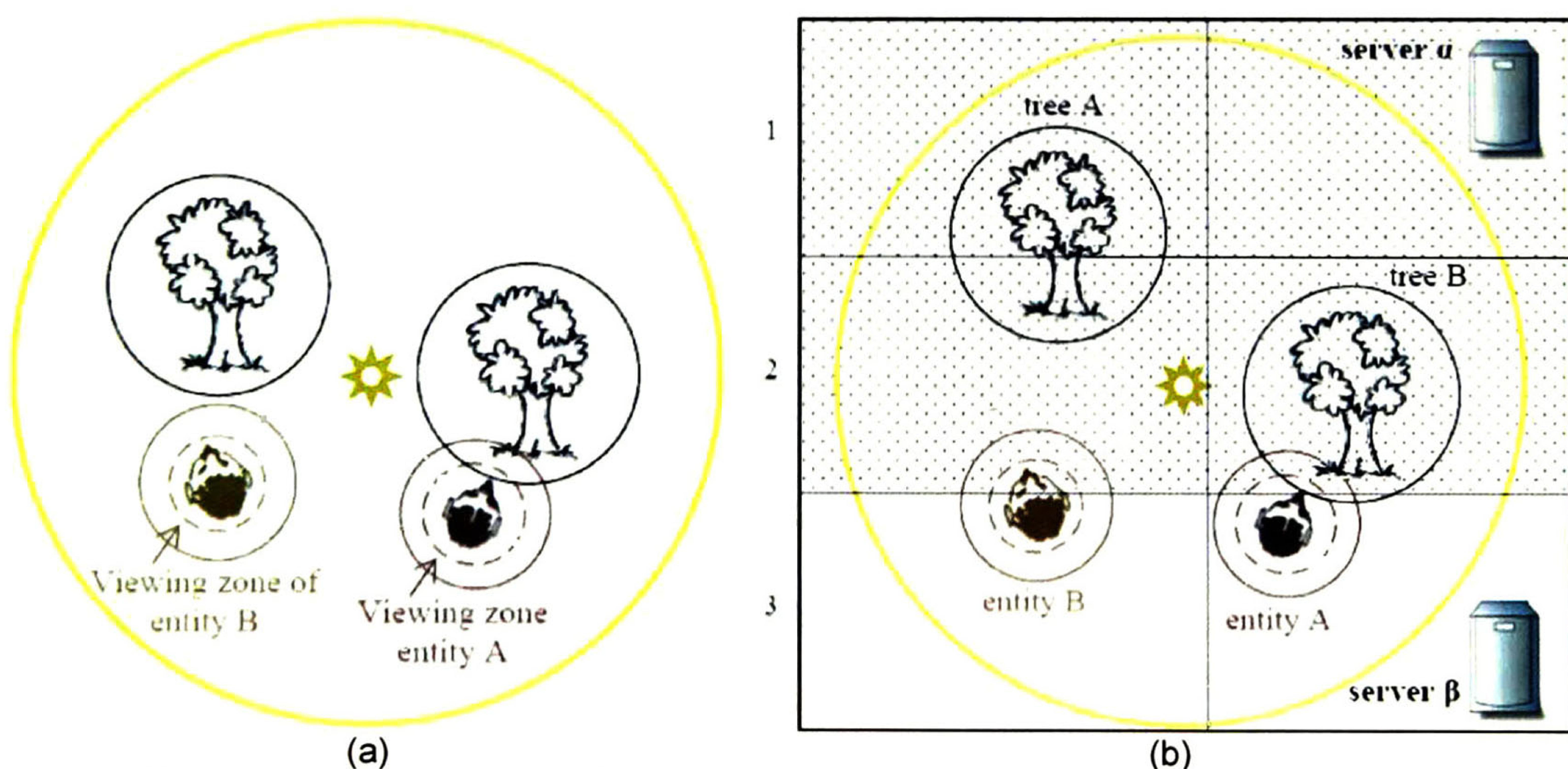
Current input perceived by agent's sensors will build a set of beliefs. Agent's behavior is determined by its beliefs. An agent will have a repertoire of actions available to it, with which it may be able to modify the environment's state. This set of actions represents the agent's effectoric capability (a set of effectors). The agent's set of current beliefs will generate desires in order to accomplish its design objectives. An action taken by an agent according to its desires is an intention.

In most domains of reasonable complexity, an agent will not have complete control over its environment. It will have at best partial control, in that it can influence it. The possibility of failure to have the desired effect in the environment leads to say that environments are in general assumed to be non-deterministic. The latter captures the fact that agents have a limited "sphere of influence".

2.3.1 Distributed Virtual Environments

Migrating from a centralized representation of the environment to a distributed one, as this thesis pursues, involves analyzing issues related to which of the entities that represent the environment, should be notified about every change done in the environment.

Inspired in biological systems [WEYNS], several researchers have shown that the environment may serve as a robust, self-revising shared memory for the agents. In FIPA (Foundation for Intelligent Physical Agents) [WEYNS] it is difficult to find any functionality for the environment further more in the matter of message transport or broker based systems. According to Jacques Ferber an environment may be represented as a monolithic system (centralized environment, Fig 2-9(a)), or as a set of assembled cells in a network (distributed environment, Fig. 2-9(b)) [ELMERHEBI].



(a) (b)

Figure 2-9. Centralized and Distributed Environments (a) centralized (b) distributed

Differences between centralized and distributed approaches are that, in a distributed approach:

1) State of a cell depends on the others that around it;
2) Perceptions of agents go further more only one cell;
3) Every time agents move from one cell to another they must update their links;
4) It must be handled signal propagation over the cell net.

DIVA, presented by [VOSINAKIS] is an example of a centralized environment that uses a world server. When an agent request an action, the world server checks its set of laws to determine whether this action will be successfully performed or not; in the first case updates the world data as necessary and notifies all visual agents that a change took place. In both cases the results are sent back to the agent client using the agent-world communication protocol.

Odell et al. [ODELL] make a distinction between physical environment and communication environment. Physical environment provides rules, laws and restrictions for entities, while communication environment provides:

1) Principles and processes to support idea interchanges, and
2) Functions and structures for communication (roles, groups and communication protocols).

Some models for interactions with the environment are:

1) Classic blackboard communication infrastructure (an intermediary data repository to communicate indirectly and anonymously);
2) Tuple based interaction model: agents in Linda (a coordination language which separates behavior, communication and synchronization) communicate placing and removing tuples from a shared space;
3) Java spaces: transfers Linda model to a distributed one offering a possible remote access to tuplespaces;
4) LIME (Linda in a Mobile Environment): for a mobile environment which instead of having a centralized tuplespace, each agent maintains its own tuplespace; the originality of this one is that, when agents reside in the same host (or a connected one) their tuplespaces mix up transparently (agents have the illusion of a tuplespace locally shared).

2.3.2 Distributed Virtual Reality Platforms

Studying already implemented platforms provides ideas based on their experience, in order to build a platform that provides analogous characteristics and also the specific purpose ones.

To get a good coherence between the states within a distributed virtual environment, whenever a change occurs at one host it must send an update message to other hosts; this process is the same in any host having a change. This process must be done quickly, because the realism depends on the good management of update messages. Besides, a participant is only interested in the part of the virtual environment that is available to him. A great number of update messages will cause traffic and may lead to a loss of messages or to an increase in latency causing problems in the realism. Thus it is crucial to integrate filtering in distributed virtual reality applications but taking care of the fact that filtering may eliminate update messages causing disappearance of important objects in the scene.

The filtering technique called the Effect Management [ELMERHEBI] associates to each object a spherical zone that reveals the importance of the object in its environment. The sphere is called the effect zone, which associates to entities (active objects) a viewing zone (visual capacity) depending on the area of interest or the aura as illustrated in previously shown Figure 2-9(a).

The latter research use Multicast because it has been proven to be a mode of communication that suits well filtering, mainly because it saves the cost of repetitive transfers reducing network traffic. Thus it could be possible to associate to each object a multicast group; when an entity overlaps an object´s effect zone the entity joins the group to receive updates, but there would be a great number of multicast groups; so they suggest dividing the space into cells and associate to each cell a multicast group. There is still the problem about the size of the cells because if they are small, entities will frequently join and leave groups (if they move quickly) on the other hand if cells are large, entities will receive data that do not interest them because of many object entities in the same area.

Thus, they propose to keep the large cells and to generate dynamic regions managed by a server consisting of one or many cells according to the number and distribution of the objects and entities. Servers will manage an intra-region filtering layer for filtering data inside its region. In this way an entity will receive from its server only the data that interest it. There is also an inter-region filtering layer where servers will communicate between them via multicast delivering the data that interest their entities; see Figure 2-9(b). As a future work they suggest a mechanism that determines the appropriate size of effect zones, the optimal size of cells and a technique that handles the dynamic regrouping of cells into regions.

The RING system described by [FUNKHOUSER] consists of client-server design and implementation for a system supporting real-time visual interaction between users in a shared 3D virtual environment. RING uses server-based algorithms to compute potential visual interactions between entities. When an entity changes its state, update messages are sent to workstations with entities that can perceive the change. The RING represents a virtual environment as a set of independent entities with a geometric description and behavior; some entities are static and others dynamic. The interacting entities send messages to announce

updates to their geometry or behavior, modifications to the shared environment or impact to other entities. Client workstations execute necessary programs to generate behavior for their entities. Clients in addition of managing their own entities maintain (simplified) surrogates for some entities managed by other clients. Communication between entities is managed by servers; clients send messages to servers which forward them to other client and server workstations managing entities that can possibly "see" the effects of the update. Prior to the multi-user simulation, the shared virtual environment is partitioned into a spatial subdivision of cells whose boundaries are comprised of the static polygons of the virtual environment as illustrated in Figure 2-10

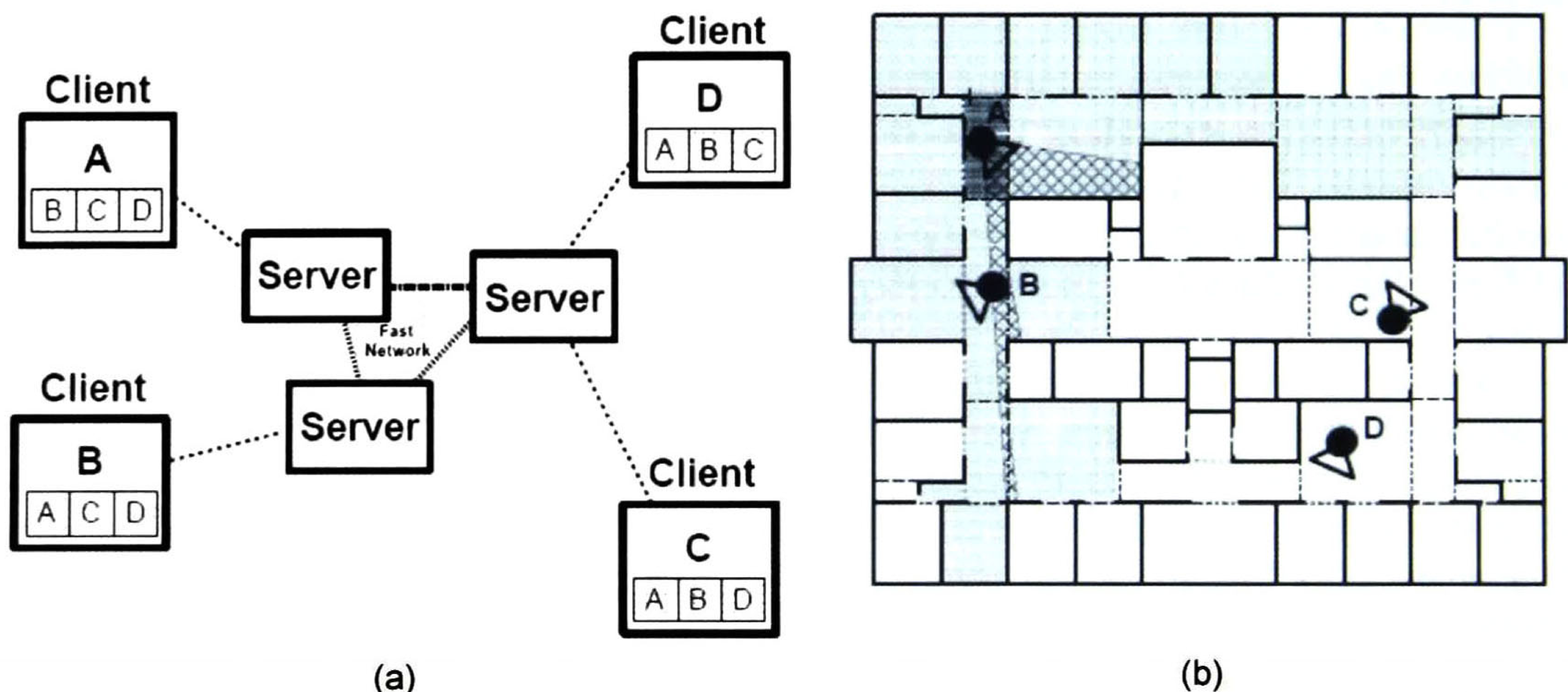


Figure 2-10. RING servers (a) RING servers managing communication. (b) Cell to cell visibility

2.3.3 Self Organization of Multiagent Systems

The realization of this thesis expects that a collection of agents, working together, represent the entire environment, decomposing the representation problem in manageable tasks efficiently. Cooperation is mandatory for the distributed environment administration.

Historically, most work on cooperative problem solving has made the *benevolence* assumption [WOOLDRIDGE]: agents in a system implicitly share a common goal, and thus there is no potential for conflict between them. The latter is acceptable if all the agents in a system are designed or "owned" by the same organization or individual. The more general area of multiagent systems has focused on the issues associated with societies of *self-interested* agents which cannot be assumed to share a common goal, as they will often be designed by different individuals or organizations in order to represent their interests. Despite the potential for conflicts of interest, the agents in a multiagent system will ultimately need to cooperate in order to achieve their goals; just as in human societies.

Task sharing takes place when a problem is decomposed to smaller sub-problems and those are allocated to different agents which may have the same or different capabilities. In cases where the agents are really autonomous and can hence decline to carry out tasks (in systems that do not enjoy that *benevolence* assumption), then task allocation will involve agents *reaching agreements* with others.

Result sharing involves agents sharing information relevant to their sub-problems. This information may be shared proactively (one agent sends another agent some information because it believes the other will be interested in it), or reactively (an agent sends another information in response to a request that was previously sent).

Contract Net Protocol

The Contract Net (CNET) protocol [SMITH77] is a high-level protocol for achieving efficient cooperation through task sharing in networks of communicating problem solvers; the basic metaphor is the contracting.

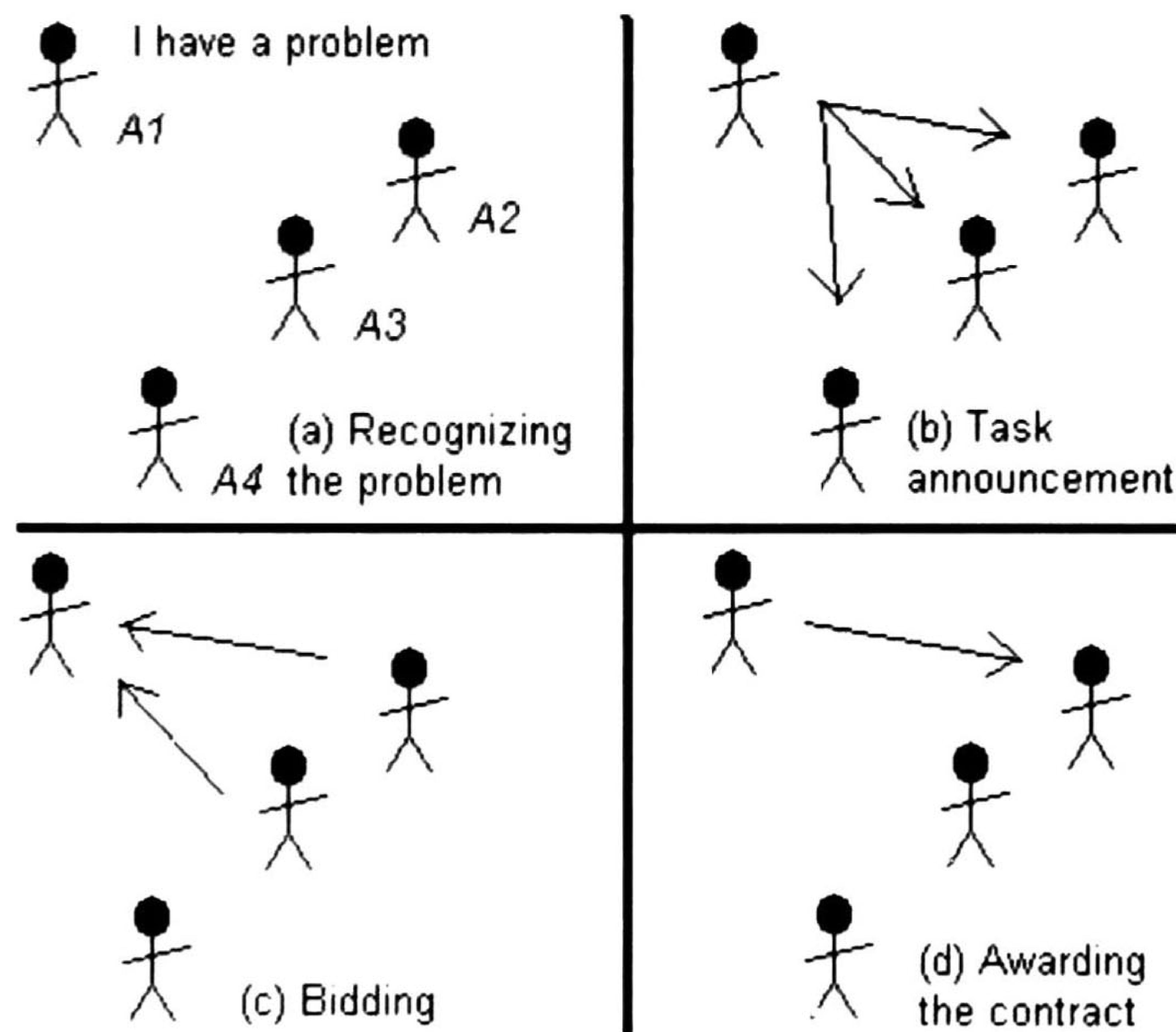


Figure 2-11. The Contract net (CNET) protocol

As illustrated in Figure 2-11, a node that generates a task advertises existence of that task to other nodes in the net with a *task announcement*, then acts as the *manager* of that task for its duration [SMITH80]. In the absence of any information about the specific capabilities of the other nodes in the net, the manager is forced to issue a *general broadcast* to all the other nodes. Depending on the information the manager knows about the capabilities of other nodes, it can issue a limited broadcast or a point-to-point announcement.

Nodes in the net listen to the task announcements and evaluate them. When a task to which a node is suited is found, the node submits a *bid*. A bid indicates the capabilities of the bidder that are relevant to the execution of the announced task. A manager may receive several bids; based on the information in the bids, it selects the most appropriate nodes to execute the task. Te selection is communicated to the successful bidders through an *award* message. These selected nodes assume responsibility for execution of the task, and each is called a *contractor* for that task.

2.4 Software Engineering

The evolution of a large software project requires that it is designed according to software engineering standards. It is a global problem that some programmers build a solution without writing documentation, and sometimes they do not even meet standards, which would help to the software life cycle. Frequently programmers do not have time to comply with it; sometimes they found an easier way, but not the best, to show expected results; sometimes we let them go before the project manager could realize missing stuff that would interfere in future. Reverse Engineering is necessary to fix this such problems. The following paragraphs describe good practices for software development, expected to be mandatory from now on for the GeDA-3D project, in order to achieve that the next generation of programmers implement their goals adequately and of course in time.

Forward engineering [CHIKOFSKY] is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system. The adjective “forward” has come to be used where it is necessary to distinguish this process from reverse engineering. Forward engineering follows a sequence of going from requirements through designing its implementation.

2.4.1 Reverse Engineering

Reverse engineering [CHIKOFSKY] is the process of analyzing a subject system to:

- identify the system´s components and their interrelationships, and
- create representations of the system at a higher level of abstraction.

Reverse engineering in and of itself does not involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of examination, not a process of change or replication.

Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered as alternative views (for example dataflow, data structure, and control flow) intended for a human audience.

Restructuring is the transformation from one representation form to another at the same relative abstraction level while preserving the subject system´s external behavior (functionality and semantics).

Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering also includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system.

2.4.2 Component Based Software Engineering

The concept of reuse in the software development context was introduced by McIlroy almost forty years ago [PACHECO]. He proposed the creation of a software component industry that would offer families of routines for any given job [MCLLROY].

While opportunistic reuse (by cut and paste of code from old systems to new ones) has been used by individuals and small teams, it does not scale up well to larger organizations and complex software systems [SCHMIDT].

Cut and paste coding is a dangerous form of reuse, in that it leads to proliferation of code clones throughout the source code. If a bug is found in a portion of code which has been reused through cut and paste, or a requirement that lead to its creation changes, then producing the required modifications in that piece of code is expensive, as it is replicated in several different clones [PACHECO]

A software *component* [SZYPERSKI] is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. The characteristic properties of a component are that it:

- is a unit of independent development;
- is a unit of third-party composition;
- has no (externally) observable state.

For a component to be independently deployable, it needs to be well separated from its environment and other components. A component encapsulates its

constituent features; also as a unit, a component will never be deployed partially. For a component to be composable with other components by such a third party, it needs to be sufficiently self contained. Finally, a component should not have any (externally) observable state so it cannot be distinguished from copies of its own.

The notions of instantiation, identity, and encapsulation lead to the notion of objects. The characteristic properties of an object are that it:

- is a unit of instantiation, it has a unique identity;
- many have state and this can be externally observable;
- encapsulates its state and behavior.

A component is likely to act through objects and therefore would normally consist of one or more classes of immutable prototype objects. A class can be seen as implementing an abstract data type (ADT), with the additional properties of inheritance and polymorphism.

The interface of a component defines the component´s access points. These points allow clients of a component, usually components themselves, to access the services provided by the component. Normally, a component will have multiple interfaces corresponding to different access points.

Coupling [LARMAN] is a measure of force with which a class is connected to others, with what it knows the others and uses them. A class with low (or weak) coupling does not depend on many others ("many others" depends on the context).

Cohesion (or, more exactly, functional cohesion) is a measure of how related and focused the responsibilities of a class are. A high cohesion characterizes classes with responsibilities tightly related so they do not perform an excessive work. A class with low cohesion does many things not related or an excessive work.

Several problems can arise when applications contain a mixture of data access code, business logic code, and presentation code [ORACLE]. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. Adding new data views often requires reimplementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods.

The Model-View-Controller (MVC) design pattern solves these problems by decoupling data access, business logic, and data presentation and user interaction as illustrated in Figure 2-12.

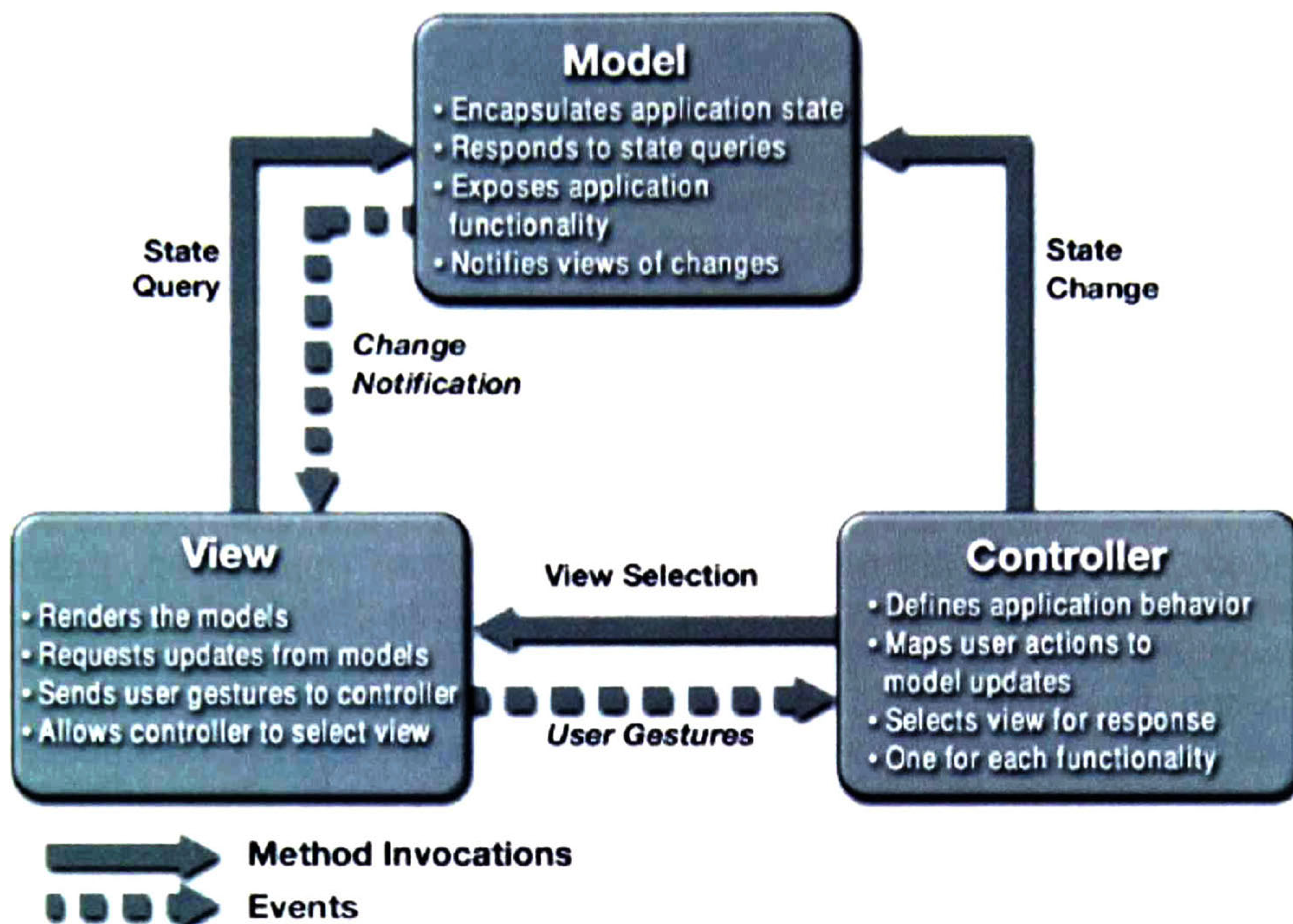


Figure 2-12. The Model-View-Controller design pattern

- Model - The model represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.
- View The view renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. Using a push model, the view registers itself with the model for change notifications, or a pull model, where the view is responsible for calling the model when it needs the most current data.
- Controller - The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

2.5 Summary

A distributed system appears as a single coherent system to its users. A middleware offers a single system view running as an application on different machines and supporting a programming abstraction. Middleware in its session layer is in charge of: providing message-passing primitives easy to use for programmers, locating processes, maintaining sent and received messages in buffers, providing reliability for message delivering, and in its presentation layer deals with heterogeneity of computers and extern data representation.

Load balancing algorithms strive to equalize workload among nodes. Load sharing policies attempt to assure that no node is idle while other components wait for service, mapping and remapping the logical system (running applications), to the physical system (interconnected processing nodes). Adaptive algorithms of load sharing comprise: information dissemination and decision-making. For information dissemination we have to decide whether to hold state information of all nodes in the system, or only a subset; the latter are used for scalability and a criteria specification is needed; a node may choose to request information, disseminate information or use a predictive analysis technique; periodic and event-driven information dissemination provide comparable performance. In decision-making the migration can be initiated by the source of work (overloaded node) or the server of work (underloaded node). With negotiation we can inhibit a bad decision. A combination of source-initiated and server-initiated policies can give the best results. About when decision making should be activated it is advocated a combination with events as the basic method and a lower bound periodicity.

When replication of data is considering mobility of servers and clients, we have six kinds of replication schemes; if the data is near to the client, reading operations are faster and this is better when reads are carried out more frequently and writes.

The minimum code migration is weak mobility, consisting in transferring only the code segment, and the program is started from one of several predefined starting positions; the only requirement is that the code is portable.

DSM consists in having a collection of workstations sharing a single paged virtual address space. If a machine does not own a page that is needed, requests the page to the owner. Some data that is needed from two or more machines might be replicated, but not all the data. The consistency of replicated common data is difficult and it is encouraged an object oriented paradigm so data is only administrated by an object which encapsulates this data and through its interfaces communicates with remote objects in order to allow data modification.

An Agent has a limited sphere of influence to modify its environment; the information perceivable through the agent´s sensors is the only that should be notified to that agent, which will evolve in its environment carrying out intentions through its effectors in order to modify the environment trying to reach its goals.

Inspired in biological systems, several researchers have shown that the environment may serve as a shared memory for the agents. According to Jacques Ferber an environment may be represented as a monolithic system, or as a set of assembled cells in a network (a distributed environment).

Odell et al. make a distinction between physical environment (rules, laws, restrictions) and communication environment (principles to support idea interchanges and functions and structures for communication such as roles, groups and protocols).

In a distributed virtual environment a great number of messages will cause traffic, and may lead to a loss of messages or to an increase of latency causing problems in the realism. Thus it is crucial to integrate filtering in distributed virtual reality applications. Multicast has been proven to be a mode of communication that suits well filtering, saving the cost of repetitive transfers reducing so network traffic.

The Contract Net protocol is a high-level protocol for efficient cooperation through task sharing. A node that recognizes a problem announces a task to other nodes and acts as a manager, nodes that are capable to carry out the task submit a bid, then the manager awards the contract to one or more selected bidders.

Reverse engineering is the process of analyzing a subject system to identify the system´s components and their interrelationships in order to create representations of the system at a higher level of abstraction. Through re-documentation we can restructure the system, preserving the external behavior. Finally, we can achieve reengineering if we also include forward engineering, through modifications with respect to new requirements not met by the original system.

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. The interface of a component defines the component´s access points; these points allow clients (other components) to access the services provided by the component.

Applications are difficult to maintain if they contain a mixture of data access code, business logic code, and presentation code; interdependencies between components cause effects whenever a change is made anywhere. The Model-View-Controller (MVC) design pattern solves these problems by decoupling data access, business logic, and data presentation.

Chapter 3

Software Engineering of GeDA-3D

In this chapter we analyze through re-documentation and reverse engineering, current components developed for the GeDA-3D project (Generic Distributed Architecture for 3D Applications). This analysis is needed to execute a reengineering of these components to integrate them with the new kernel that supports the requirements for the distributed environment.

3.1 Introduction

The ultimate goal of the GeDA-3D project is to facilitate to inexperienced users the description of virtual environments using a similar to natural language. These virtual environments are intended to develop simulations based on behaviors according to distributed artificial intelligence. Examples of simulations are: urban traffic control, fire fighting, among many others. By a declarative description, a user currently is able to:

1) Create a virtual environment,
2) Specify the involved agents' attributes,
3) Assign personalities to virtual entities,
4) Specify goals to be fulfilled by the entities, and
5) Specify initial locations for every entity in the virtual world.

Zúñiga [ZUÑIGA] and Aguirre [AGUIRRE] studied various multiagent platforms such as Jade, Zeus, FIPA-OS, OMG MASIF Open Agent Architecture and others. They determined that such existent works did not satisfy completely the requirements needed for developing 3D applications based on agents. The current requirements for the middleware are the following:

1) Completely distributed execution;
2) Open source;
3) FIPA compliant architecture;
4) Easy to use and extend for programmers and non programmer users;
5) Agent replication and mobility capabilities;
6) Complete and dynamic agent administration (creation, starting, stopping, addition at runtime);
7) Agent goals setup;
8) Agent skills assignment;
9) Private agent knowledge;
10) Human-agent interaction;
11) Assignment of personality and emotional state to agents;
12) Environment simulation;
13) Multiple environment executing concurrently;
14) Agent assignment to each avatar (a graphical virtual entity);
15) Avatars database administration;
16) Environment-Render direct interaction;
17) Virtual environment collision event handling;
18) Unified shape description of agents between render and agent private knowledge;
19) Sensors and effectors simulation;
20) Transparent communication between all modules and all entities;
21) Scene evolution control, validating each agent intention to change virtual world.

The platforms that approximate to these requirements (where Jade is the closest one) should be restructured for compliance. That was the reason that led to the development of a platform that would provide the previously listed requirements.

Foremost, it is necessary to unify all descriptions from authors [RAMOS, PIZA, ZUÑIGA, AGUIRRE and MARTINEZ], about the components involved in the GeDA-3D project in a single document; after that, we are going to analyze the work done, what is expected to achieve, and finally, how to contribute to the reaching of the ultimate goal.

The contribution that we are expecting to achieve is the implementation of a distributed environment. The following review is necessary due to the current architectural design and implementation of GeDA-3D cannot support the integration of a distributed environment. The environment agents will be in charge of distributing the representation of the environment across computers and, also distributing workload for all validations required during the evolution of a scene.

3.2 Redocumentation of GeDA-3D

The GeDA-3D project starts with the design of the Virtual Scene Creator Architecture (ViSCA) proposed by Piza [PIZA] as illustrated in Figure 3-1.

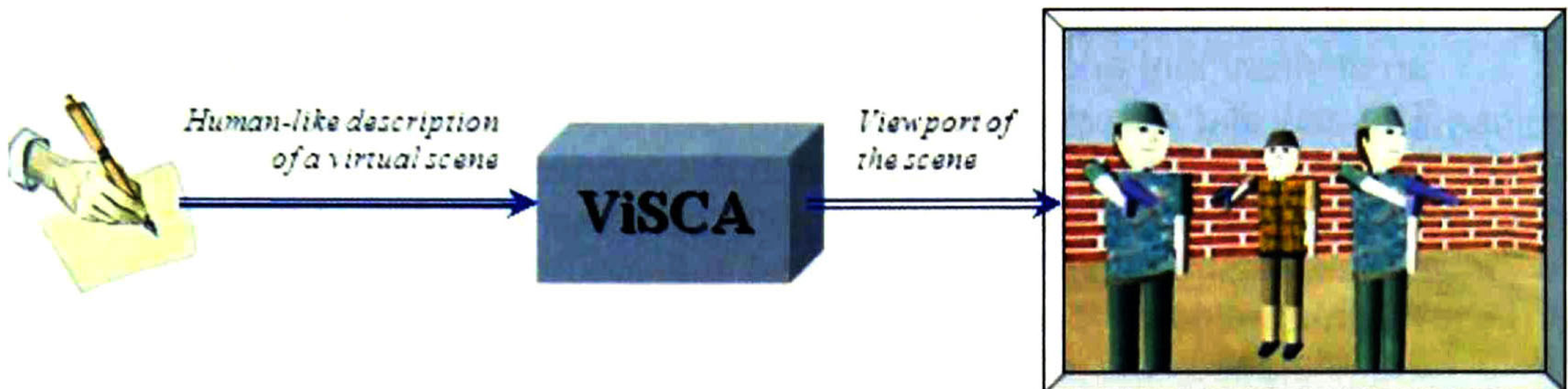


Figure 3-1. ViSCA seen as a black box

The GeDA-3D architecture has gotten improvements since the first version, and the resulting work done by [ZUÑIGA] and [AGUIRRE] is illustrated in Figure 3.2.

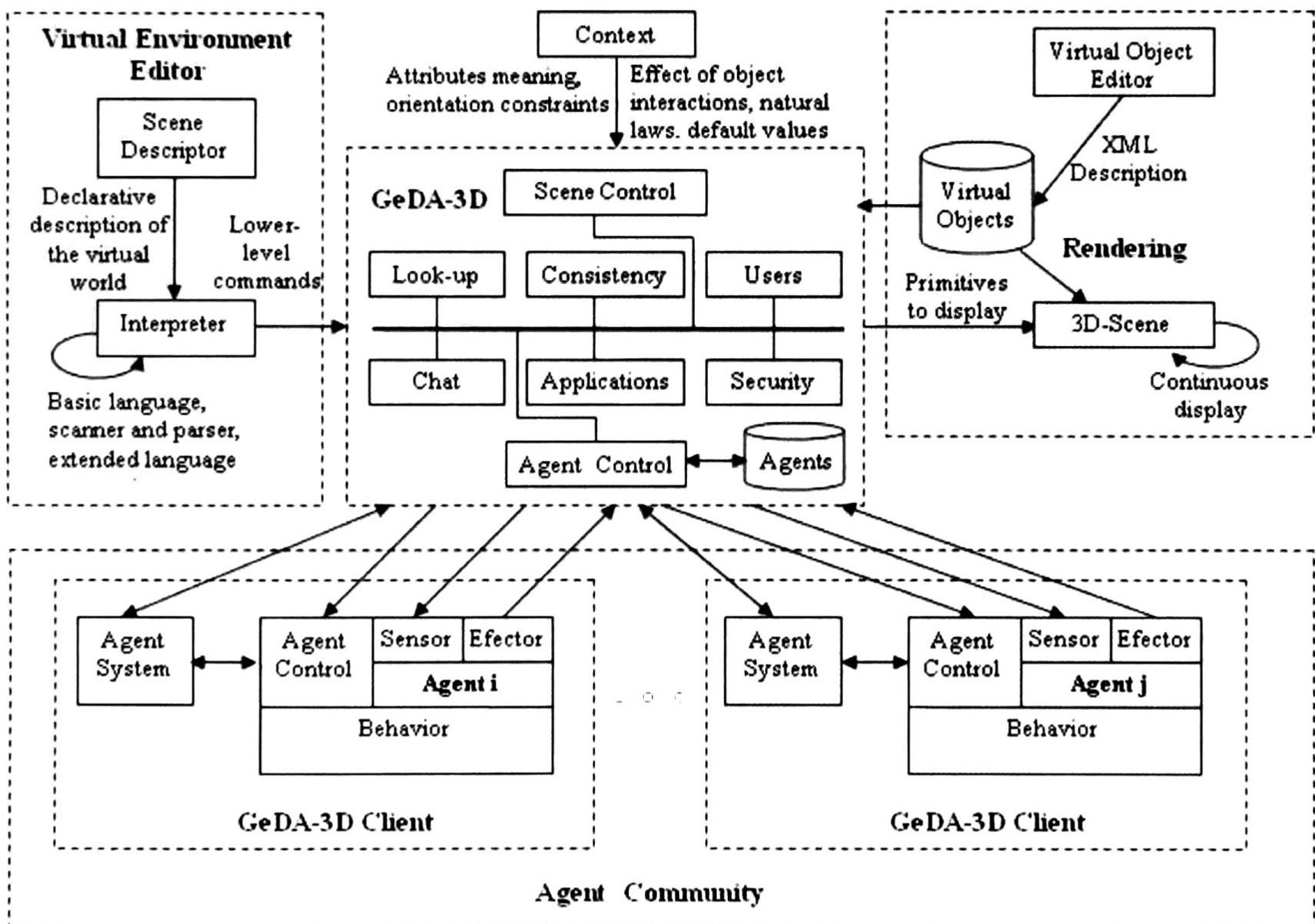


Figure 3-2. Initial GeDA-3D architecture

The GeDA-3D components are described below.

3.2.1 The Context Module

The Context gives sense to a world or environment by means of a set of definitions. Some of these definitions concern to:

- The default display of the scene, that is, the virtual entities that characterize an environment and that will appear since the beginning of the evolution;
- The set of rules restricting the execution of actions or the assignment of goals to characters, if they are not congruent to the capacities of the characters. For instance a worm cannot jump;
- All valid avatars that can be involved in a scene;
- The natural laws ruling the interactions taking place inside the environment. For instance, the effect of collisions (intended or not expected) between virtual objects, the effect of the execution of an action involving one or more objects, gravity and restriction of areas for some objects;
- The definition of all valid skills that can be included in a goal-specification;

The context is based on ontologies, which determine permitted actions according to preconditions and post-conditions, so it defines rules for the scene evolution and consequences in determined circumstances.

3.2.2 Virtual Environment Editor

The Virtual Environment Editor serves as an interface between the platform and the user. It provides means to a modeler to specify the physical laws governing an environment and, to describe a virtual scene taking place in such environment. This editor allows a user trough a declarative language to:

1) Create a virtual environment;
2) Specify the attributes of the virtual entities or agents involved in the described environment;
3) Assign personalities to virtual entities,
4) Specify goals to be fulfilled by the entities,
5) Specify initial locations for every entity in the virtual world

The editor also performs a lexical, syntactic and semantic analysis of a given description in order to translate the high-level description into a set of lower-level commands which will be sent to the congruency analyzer to verify if the scene description is correct or not. If a scene is valid, commands are sent to the Scene Control.

3.2.2.1 Scene Descriptor

It allows users to easily describe virtual scenes with the help of a visual interface and a declarative language. It provides to scenarists high-level means to create virtual entities, assign goals and specify interaction rules as illustrated in Figure 3-3.

The language used provides a set of reserved words with a predefined meaning and usage. The scene description may also contain user-defined words, such as identifiers, virtual object names, behavior names, attributes, intentions and orientation constraints. The syntax of the language defines a number of productions, which rule the way all these words can be combined together. The language can be extended in order to allow using of a wider set of words more appropriate to the kind of environment to be modeled, that is, it allows the definition of more productions in terms of the existing ones.

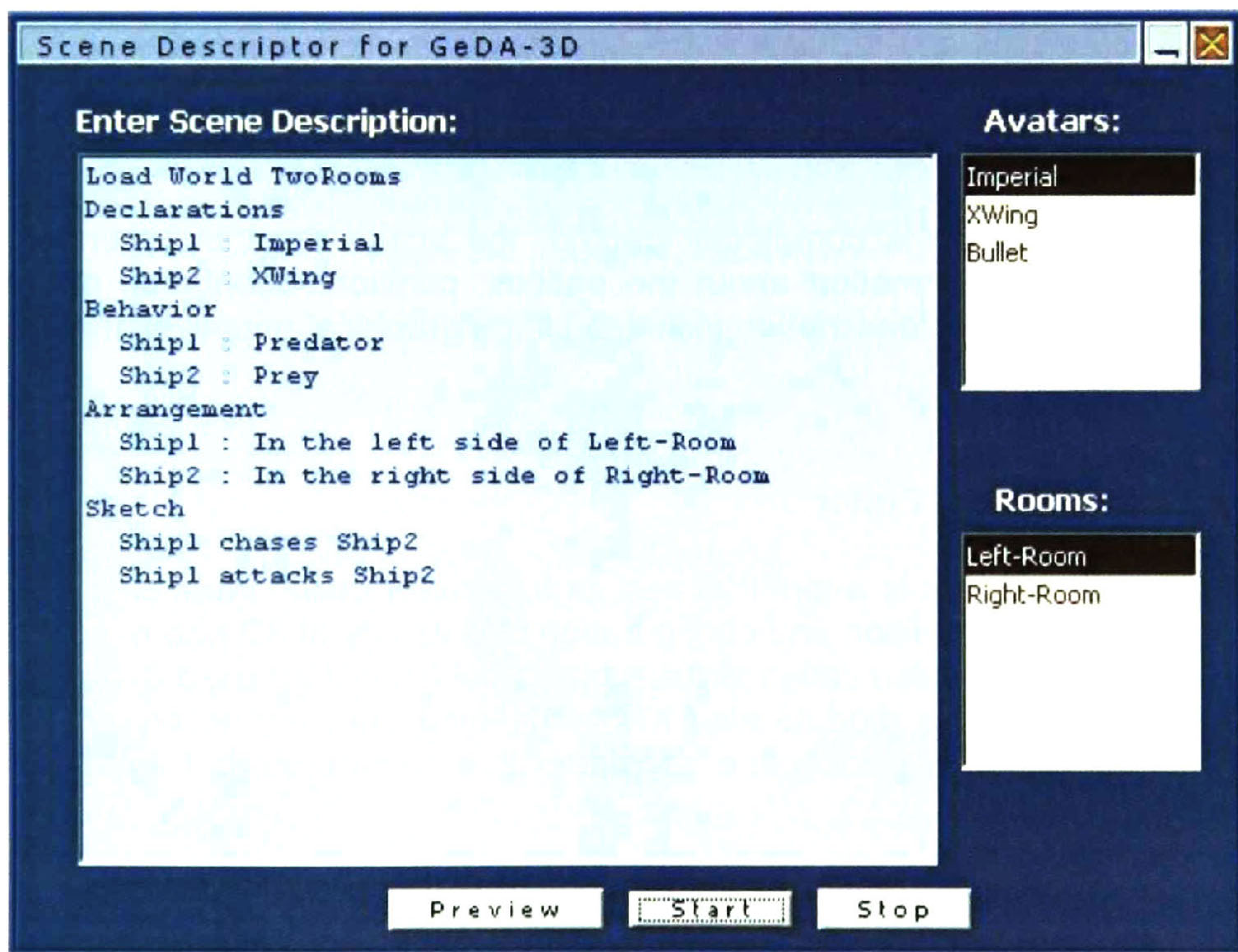


Figure 3-3. Scene Descriptor

3.2.2.2 Interpreter

The interpreter performs the lexical, syntactic and semantic analysis of a given description in order to translate the high-level description into a set of lower-level commands which are sent to the scene control. Semantic rules are defined within the context. If the language was extended in the context, part of the lexical and

syntactic analysis is performed using the rules defined in the context. The interpreter includes a constraint solver in charge of translating all found geometric constraints to 3D coordinates. It resolves the actual position of the objects in the case where the modeler provides orientation constraints.

If the scene description is error-free, the interpreter creates an XML-based description which may contain information to:

1) Link predefined virtual objects to the platform and add them to the scenario;
2) Specify the attributes of the virtual objects;
3) Allocate a behavior to every active virtual object. Behavior algorithms are defined in agents previously linked to the platform;
4) Display virtual objects in a certain state, position and orientation;
5) Send the agents the goals they will try to accomplish.

3.2.3 Rendering

Rendering addresses all the issues related to 3D-graphics. It allows the design of virtual objects and the display of the scene.

Once a description is completely parsed, the scene descriptor sends to the rendering module information about the entities: position, orientation and avatar description. The avatar description includes all the graphical primitives that build up a virtual entity.

3.2.3.1 Virtual Object Editor

Virtual Object Editor is a tool that assists to visually build virtual characters –or creatures– from the addition and configuration of a variety of 3D geometric shapes. These characters are also called virtual objects, which will be used eventually in a scene description. This module also allows defining primitive actions in terms of change of states. The architecture considers this component but it is still under construction.

3.2.3.2 The 3D-Scene

The 3D-Scene provides a view port to display all graphical changes taking place in the environment. It allows users to navigate inside the graphical scenario. It participates in the scene evolution determining collision events within the environment and informing about them to agents. Also, for every object which state is modified, this module sends the current state to agents.

This module is in charge of drawing continuously the scene. During a scene, the module keeps receiving action requests from Scene Control.

Two modules have been tested, the one developed by [PIZA] (see Figure 3-4) and the developed by [MARTINEZ] (see Figure 3-5).

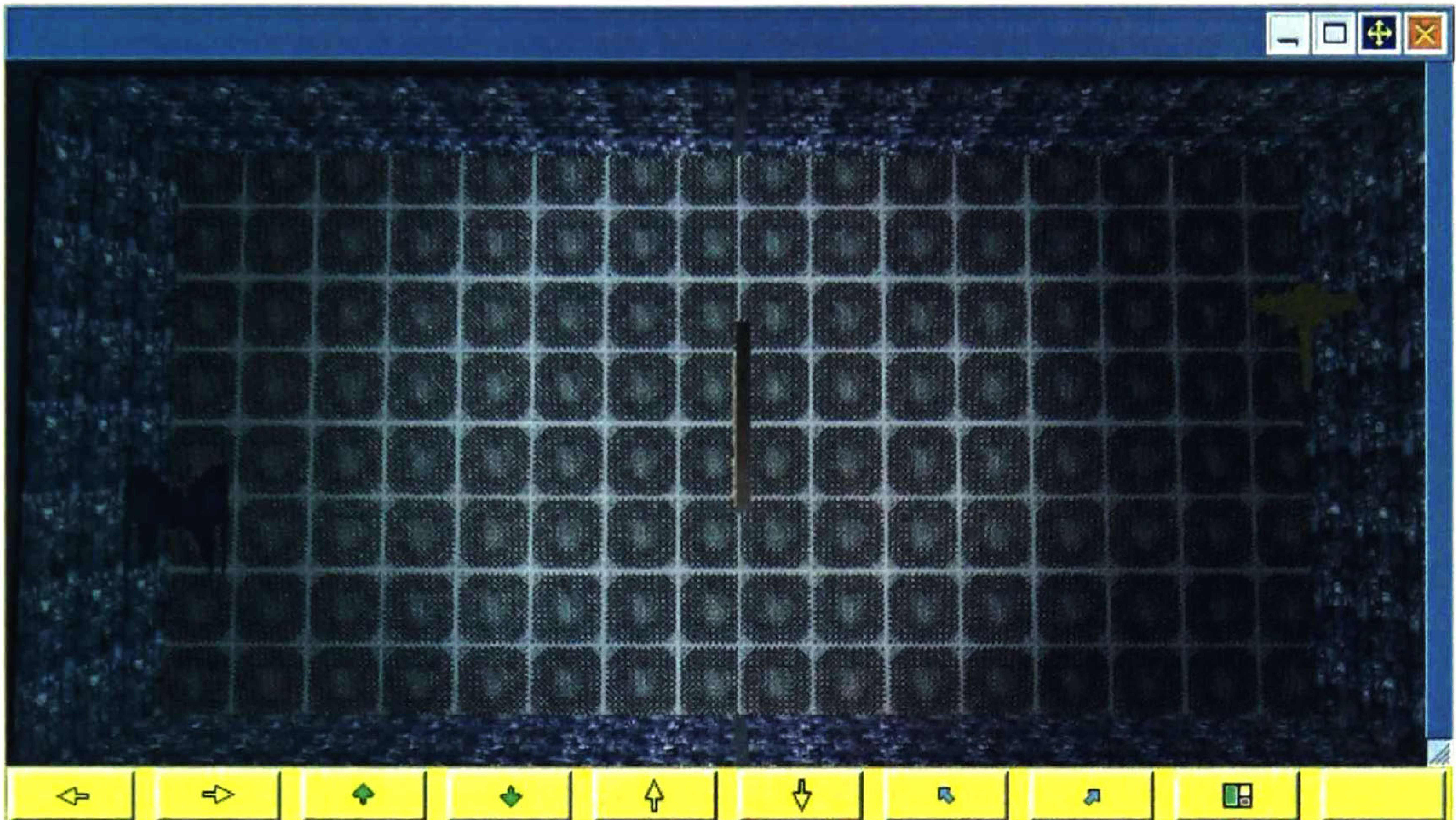
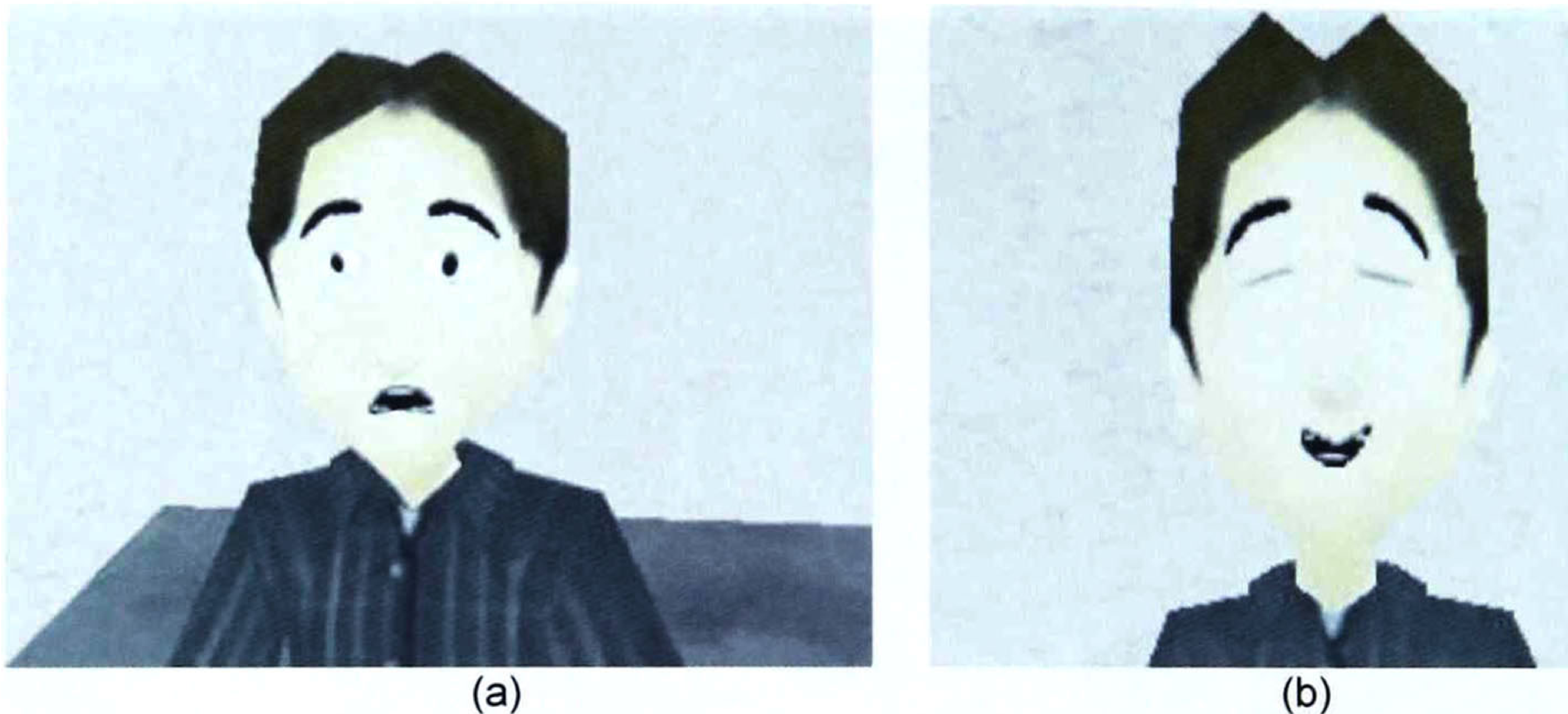


Figure 3-4. Display of ViSCA Render

Figure 3-5. Display of AVE-3D Render

The 3D-Scene also provides the mean of displaying emotional state of characters as shown in Figure 3-6.



(a) (b)
Figure 3-6. Graphical emotional state. (a) Fear (b) Happiness

3.2.4 Scene Control

It addresses all the issues related to the evolution of the scene according to the interaction of the living virtual objects. During the scene, agents suggest the execution of many actions, but virtual characters will not always be allowed to start execution of every suggested action. According to the interactions rules, Scene Control determines which actions can be carried out at certain moment (pre-conditions) and which actions should be launched as a side-effect (post-conditions). The Scene Control performs the following tasks:

1) Receives a sequence of commands in XML format for the display of the virtual objects;
2) Receives actions from active agents;
3) Validates in context the execution of each primitive action according to the world rules and obtains the effect; it has the capability to cancel the execution of a single action or an action succession;
4) Manages the generic natural laws and the context-specific ones;
5) Translates the specification of the environment into a set of primitive commands used during the rendering; such commands may be translated to LIA [MARTINEZ] or another one;

Whenever a primitive action succession is finished (successfully or prematurely), the scene control sends to the Render the required changes in the correct order. An event launcher informs every goal fulfilled and event around that occurred.

The Scene Control manages the scene evolution according to an Action-Reaction model [AGUIRRE]. This model starts informing the current state of the

environment, then agents compute actions that are collected by the scene control, it proceeds to validate actions and generating the corresponding reactions; once actions and reactions are executed, all changes are informed to agents; see Figure 3-7.

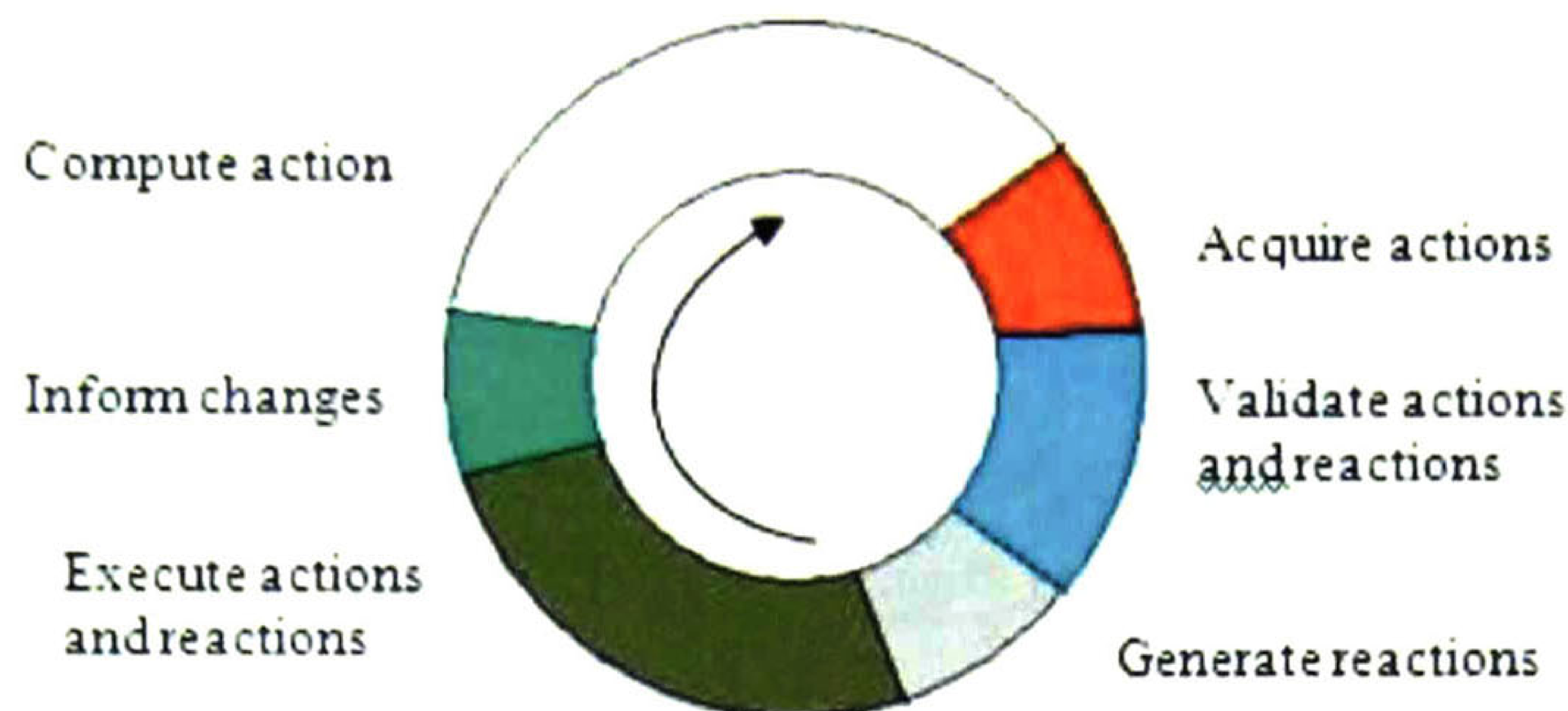


Figure 3-7. Action-Reaction cycle in the Scene Control

3.2.5 Agent Community

An agent is an autonomous process running algorithms to implement the behavior of a virtual character. Every agent controls the behavior of an active virtual entity to accomplish a goal-specification defined in the declarative description according to its defined skills.

An agent receives the goal-specification assigned to its character, after that, it resolves a set of primitive actions in order to fulfill the current goal and finally sends to the environment such actions expecting to modify the environment state.

The environment is represented inside the scene control. The agents have sensors and effectors being these the interface between the agent and the environment. Agents receive the environment update through the agents' sensors, and the agents perform actions through their effectors. Effectors send such actions to the scene control in order to modify the environment.

The Agent Architecture illustrated in Figure 3-8 is the base for the development of the Agents Community. Such agent architecture satisfies all needed requirements to allow the agents behave in environments defined through a declarative description. The architecture satisfies personality simulation, emotions, goals administration, knowledge base and shape notion of the agent.

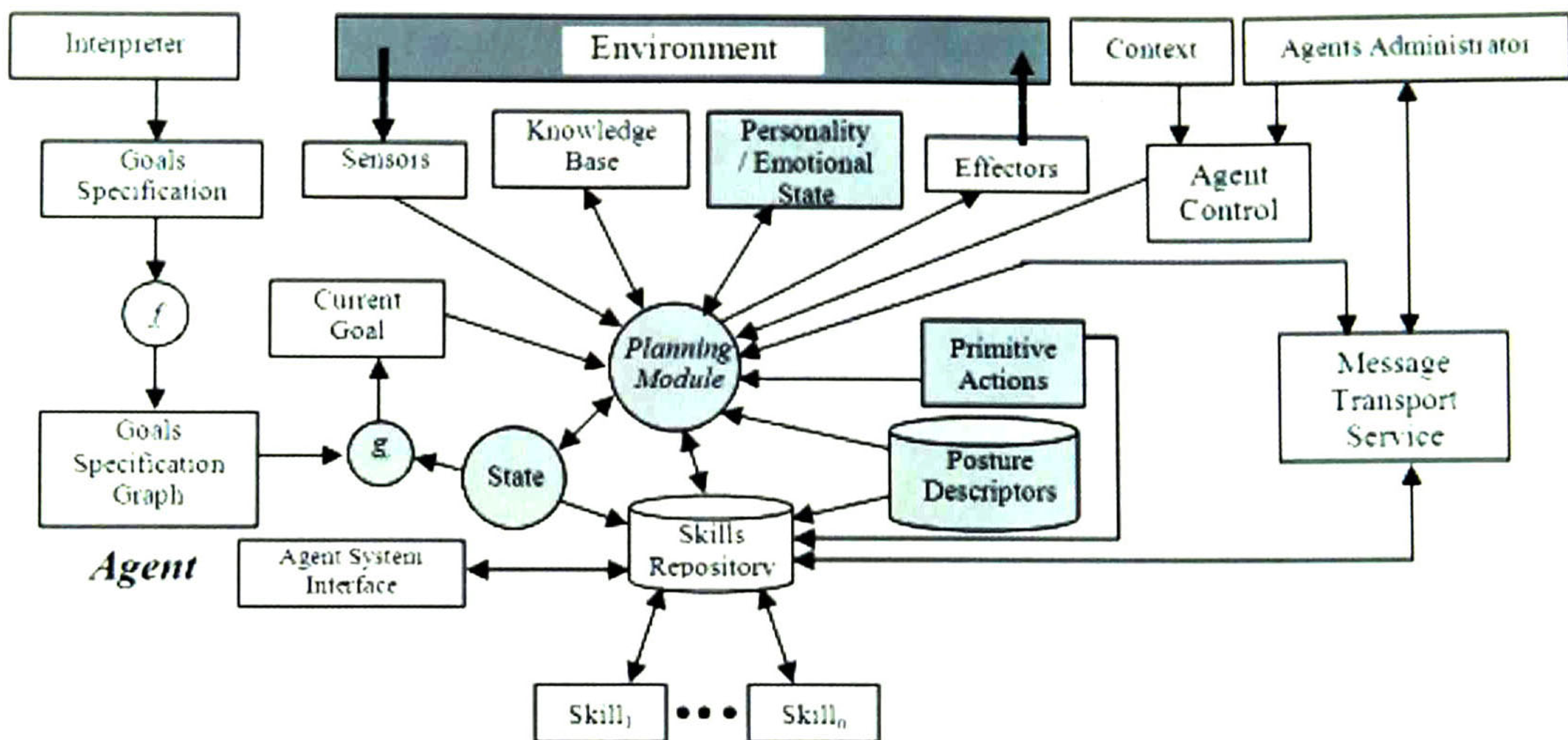


Figure 3-8. GeDA-3D Agent Architecture

An implementation of the Agent Architecture was provided to develop the behaviors of the agents described in the declarative description. When the end user intends to implement a new agent, most of the features stated in the Agent Architecture are already available like:

- Communicating with any agent or service of GeDA-3D;
- Receiving and handling automatically goals specifications at execution time;
- Receiving automatically environment representation updates according to the sensors defined by the agent;
- Executing actions over the environment through its effectors;
- Using any skill located stored by the Agent Administrator. Only the skills defined to the agent at the declarative description are activated at execution time. Skills are detailed in [ZUÑIGA].

The interaction between all the modules previously described is illustrated in Figure 3-9.

3.2.6 Agent Platform

The GeDA-3D platform is required to be based on FIPA Agent Abstract Architecture [FIPA], which is explicitly neutral about how services are implemented; see Figure 3-10.

According to [AGUIRRE], the decision about the kind of middleware to be implemented is a message-oriented middleware [EMMERICH]. This kind facilitates message exchange, supports asynchronous message delivering very naturally and also group communication. The message can be a notification about an event or a request for a service execution from a server component. The client continues

processing as soon as the middleware has taken the message. The server responds to a client request with a reply-message containing the result.

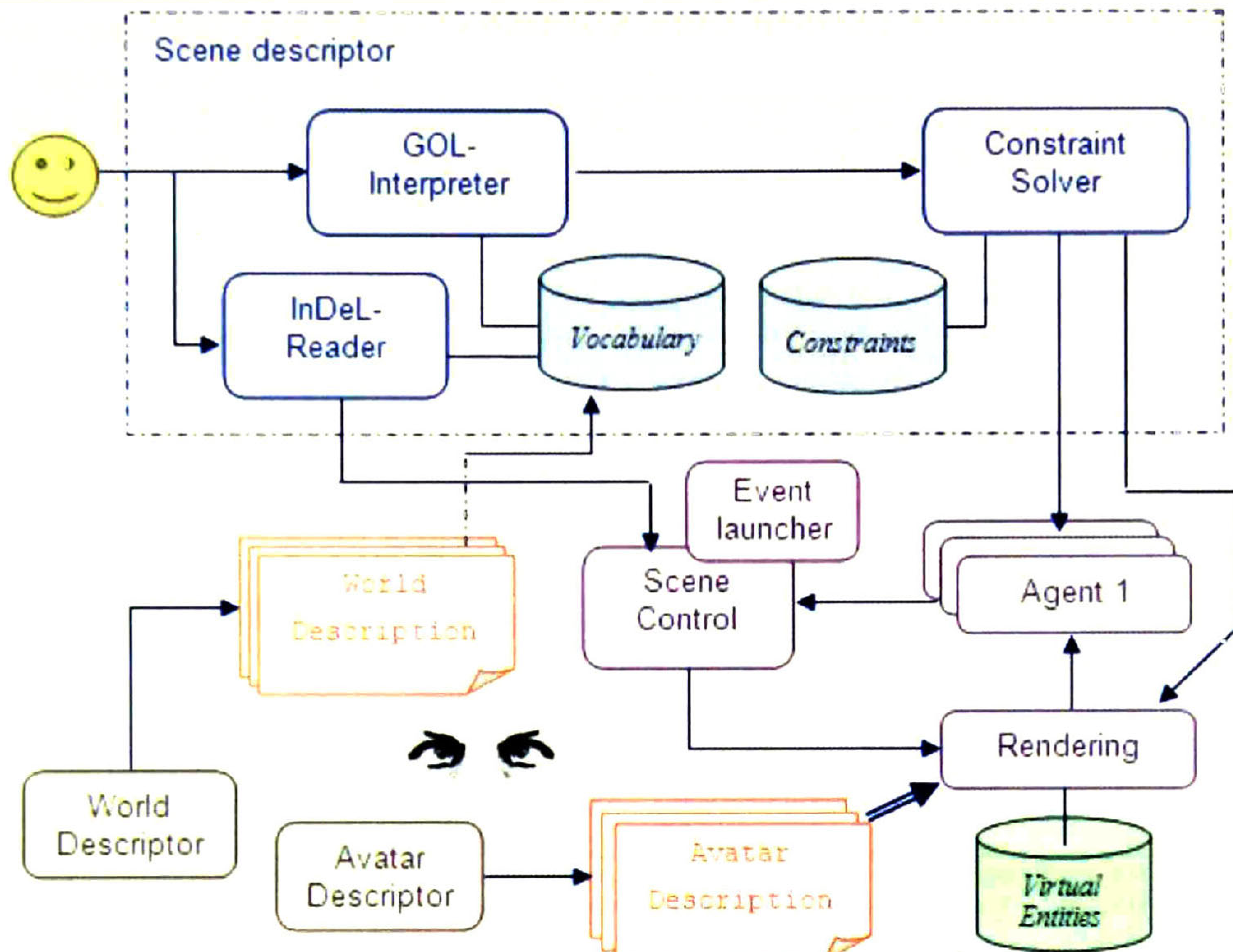


Figure 3-9. Virtual Scene Creator Architecture (ViSCA)

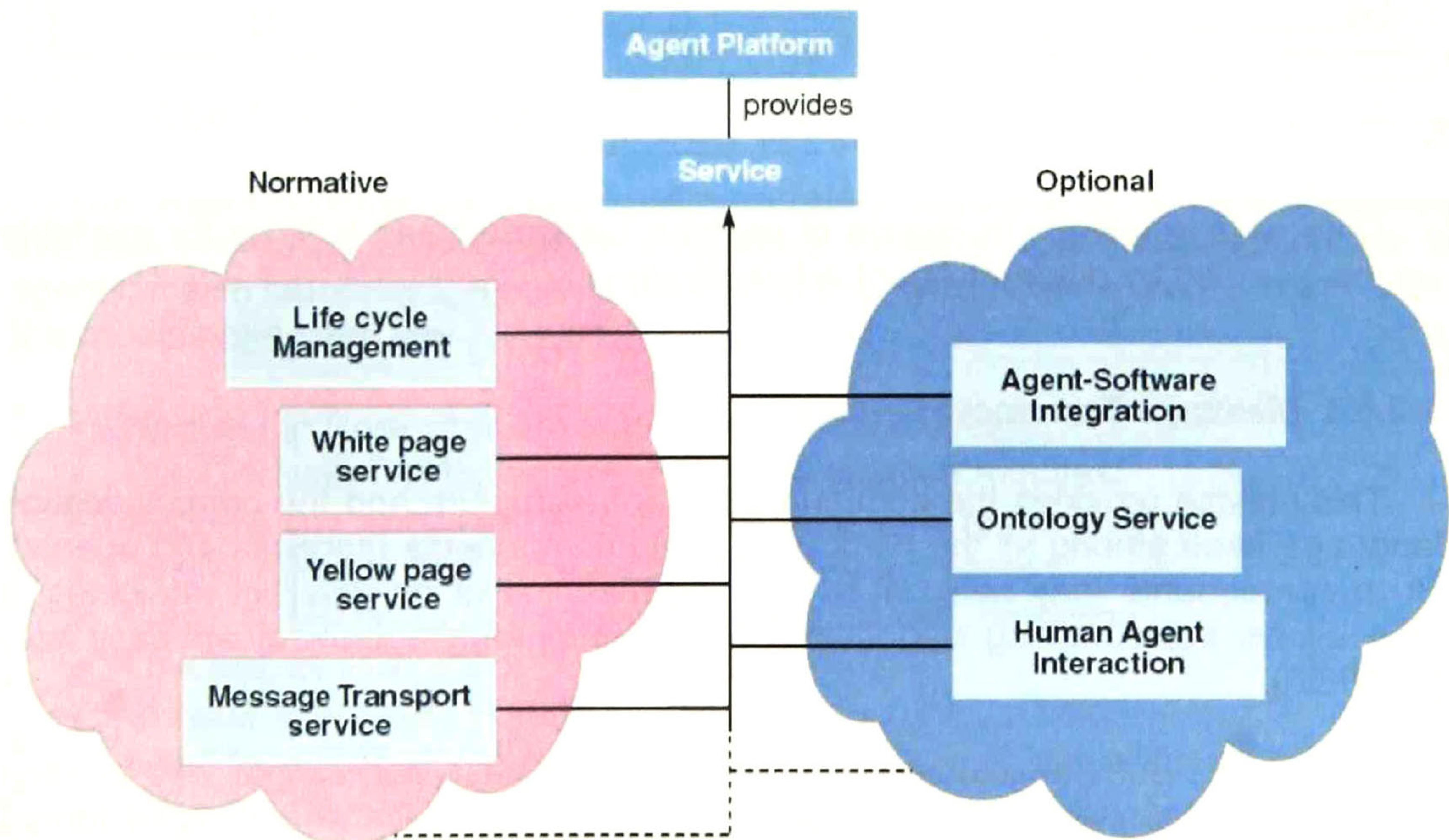


Figure 3-10. FIPA Agent Abstract Architecture

The GeDA-3D Core [AGUIRRE] is an integrator and controller of modules, it defines the way the agents interact to achieve their goals; it is the responsible to deal with heterogeneity, providing a development environment for all modules. The graphical interface of the platform, Core, is shown in Figure 3-11.

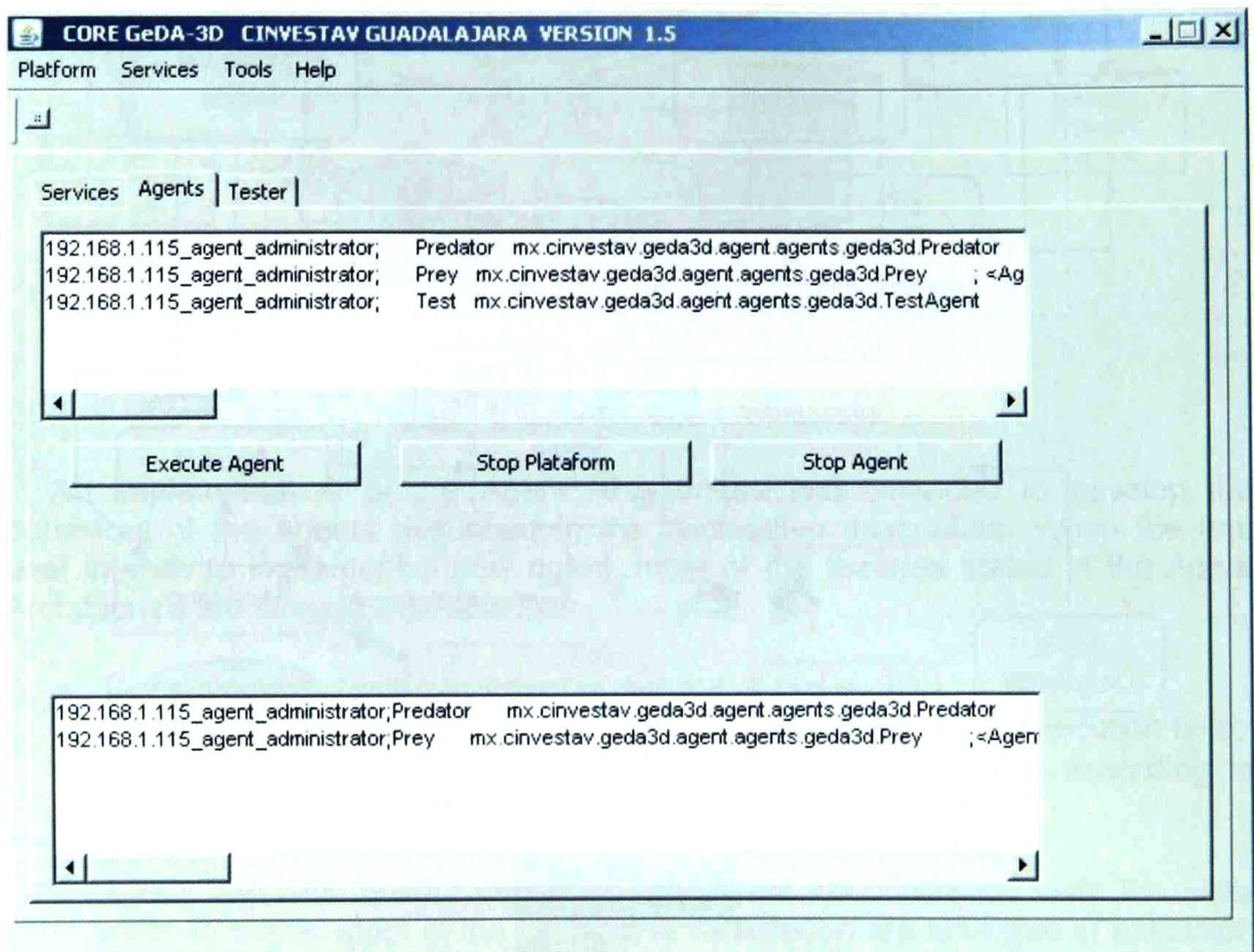


Figure 3-11. GeDA-3D Core

Core as a service is in charge of register, unregister and look up for available agents handled by different Agent Administrators.

3.2.6.1 Message Transport Service

This service provides the communication infrastructure and the communication language used among all the platform running components (services and agents). Such components may be local to a single machine or on different machines. It also allows implementing negotiation protocols among agents in an easy way [ZUÑIGA].

It hides the communication issues providing an easy and simple way to send and receive messages. The messages are self-contained; the high level format used lets precise contents specification, with which there is a minimum invasion to

the autonomy of each component in the system, facilitating the openness of the project [AGUIRRE].

The messages contain two parts: the header and the ACLMessage [ZUÑIGA]. The header of the message includes transport information used to deliver the message to the right destination. The rest of the message represents the information to communicate and it is specified as XML format coding a FIPA ACL message. ACL stands for Agent Communication Language [FIPA_SPEC].

A FIPA ACL message contains a set of one or more message parameters. The parameters needed for effective agent communication will vary according to the situation. Specific implementations are free to include user-defined message parameters other than the FIPA ACL message parameters. The Message Transport Service implements the FIPA ACL message parameters allowing the user to set user-defined ones. Agent Message Transport service allows agents to:

- Send messages to other agents;
- Receive messages from other agents;
- Broadcast;
- Include binary content (useful to send resources like files);
- Send encrypted messages;
- Receive encrypted messages;
- Encrypted Broadcast.

Agents do not need to check if they have received a message. The message is delivered to the corresponding agent just when the message arrives.

3.2.6.2 Agent Control Service

The Agent Control (or Agent Administrator) registers in a local database available agents for environments; it registers available agent skills attachable to agents; it also launches the agents needed in the environments. Some of the tasks this module executes are:

- Register in Core all agent definitions available locally;
- Start (activate) all the agents a scene simulation requires;
- Receive and assign the described internal state to the agents;
- Assign the skills and personality described to agents;
- Lets add agents to its database, reading an XML definition;
- Link agents to avatars;
- Consult agent definitions;
- Send to agents ACL messages to achieve a human-agent interaction;
- May send a message to stop execution of an agent at any time.

All tasks this module performs use an Agent Control interface inside every agent. The graphical interface of the Agent Administrator is shown in Figure 3-12; the agent definition is shown Figure 3-13; and the mechanism to interact with the agent is shown in Figure 3-14.

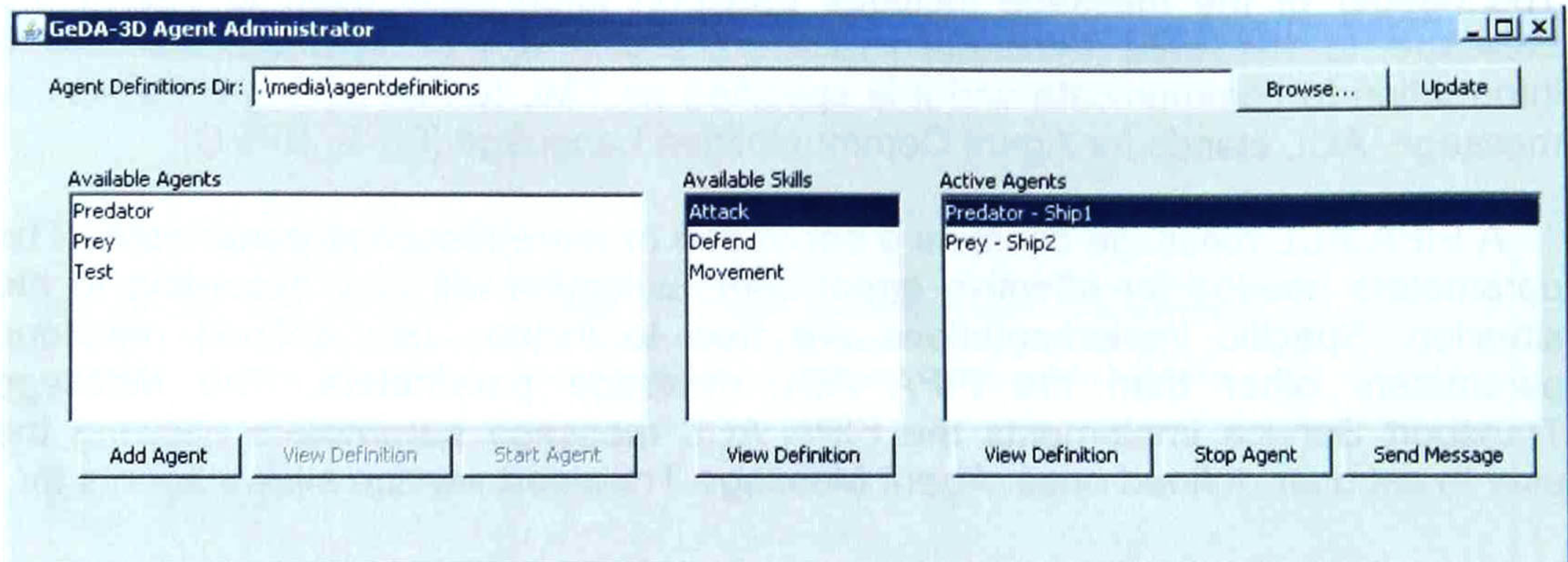


Figure 3-12. Agent Administrator

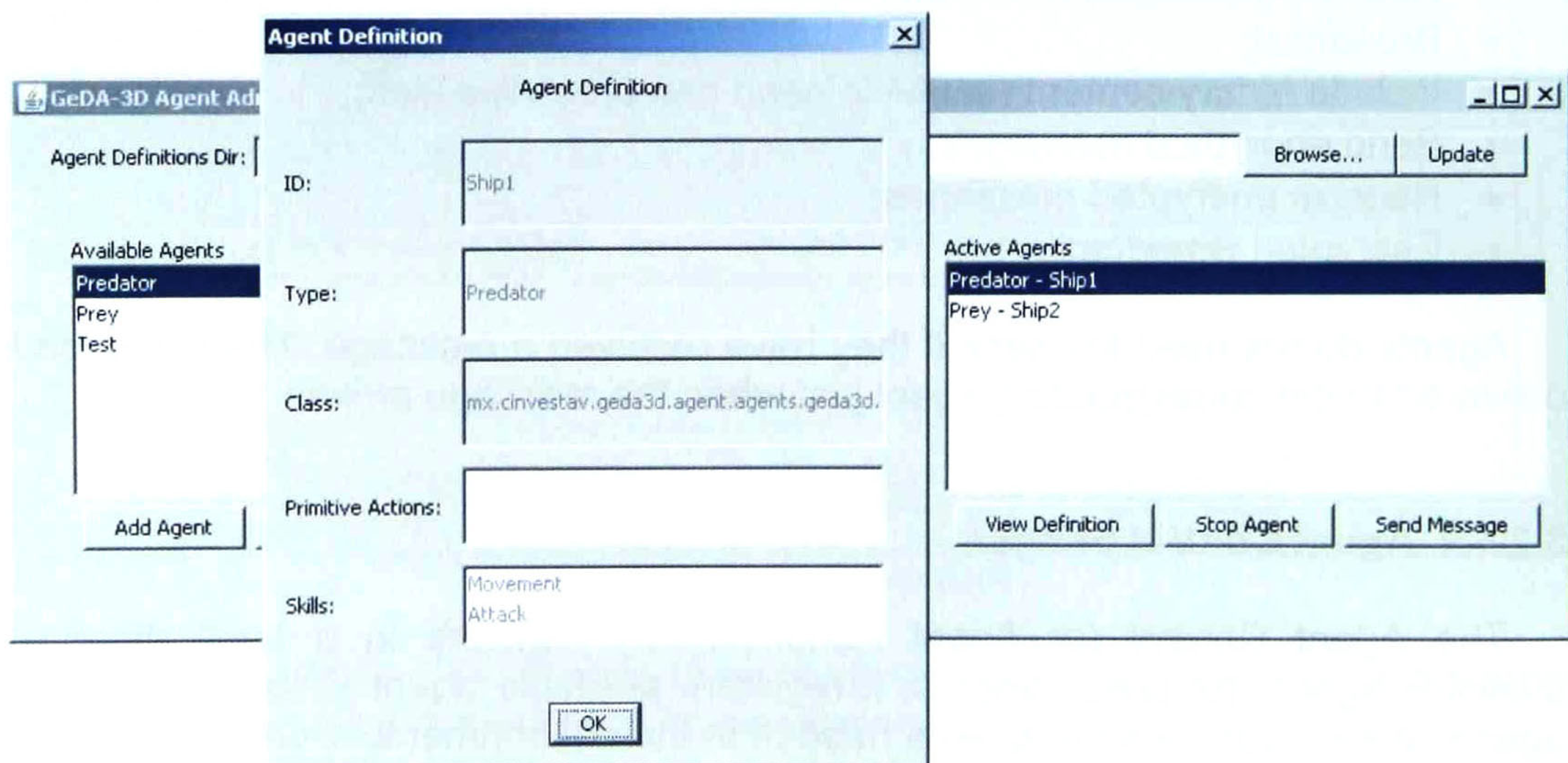


Figure 3-13. Agent definition view

3.2.6.3 Other Services

ACL/LIA-3D conversion service: transforms ACL commands to its corresponding LIA-3D representation [MARTINEZ]. Permitted actions by LIA-3D are determined on a database of available actions per avatar; this database is inside the AVE-3D module.

Look-up service: this service, stated in GeDA-3D architecture is in fact, the Message Transport Service.

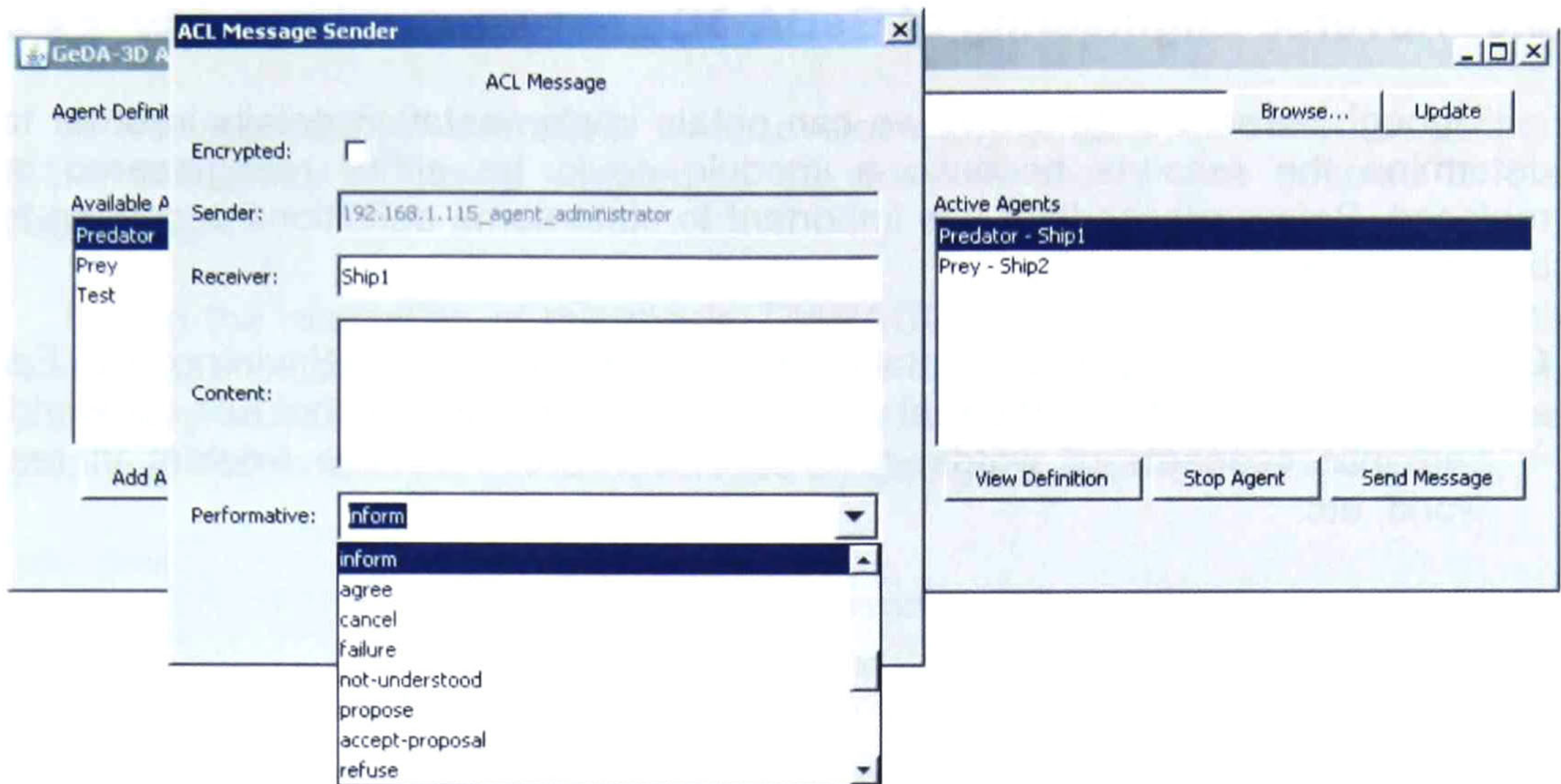


Figure 3-14. Human-Agent interaction

Security service: this service is included in Message Transport Service giving a choice to send, receive, or broadcast an encrypted message.

Mobility Service: this service could allow execution of a process in several execution environments. It could be used to share skills among agents, where such skills would be able to be executed remotely or to be moved to the machine where the requested agent is being executed.

Knowledge Base: it is used by agents to provide them with an initial knowledge of the world. This allows developing behaviors focused on the goals specification fulfillment avoiding time spent recognizing the world before trying to accomplish the goals specification. An agent could use private and/or group knowledge; for the latter the agent would have to subscribe to a group in order to have access to shared knowledge.

Resource administration service: some modules require various resources stored local or remotely. Resource registration could be a way to publish available platform resources in the system; when a node receives a request, sends back the required resource. So, modules may share resources like files, classes, world definitions, images, etc.

Finally, the services Consistency, Users, Chat and Applications have not been implemented neither formally defined.

3.3 Reverse Engineering of GeDA-3D

Through reverse engineering we can obtain implementation details in order to determine the reasons because a module could be either reengineered or replaced. Before proceeding, it is important to state some definitions according to our approach:

Context: is the entity that contains information about the Environment. For example, the context in the real world contains physical laws that rule our world, semantic concepts of words, relationships between entities existent in real world, etc.

Scenario: is the medium where a scene is carried out.

In our approach, the Environment will be represented by a Scenario and a Context together.

3.3.1 The Context Module

A first approach of this module was implemented by [PIZA]. Such Context was not implemented as an autonomous module as stated in GeDA-3D initial architecture (see Figure 3-2). The goal of Piza was satisfied, but unfortunately, the implementation of this module was not made to be a software component with an interface (see section 2.4.2 about component based software engineering).

The Context developed by Piza is strong coupled to the Scene Descriptor designed also by Piza, also does not comply with the MVC design pattern. It is very difficult to separate that context for future tests. The designing of a new one is necessary, using of course the ideas proposed by Piza.

A second approach of the Context was proposed by [ZARAGOZA] through a module called Context Descriptor, which is used by the Virtual Environment Editor also developed by Zaragoza. The Context Descriptor evaluates the scenario described by the end user and if the Descriptor determines that is a valid scene then sends it to the Rendering module.

The Context developed by Zaragoza does not comply with the MVC design pattern because the source code is coupled with the editor. It will take some time to determine how to separate the corresponding java classes in order to comply with the GeDA-3D architecture.

The stated responsibilities of the Context Module (see section 3.2.1) and, the contribution this thesis pursues, the dynamic distribution of the Scenario and the Context, lead to the forward engineer described in section 3.4.

3.3.2 Virtual Environment Editor

A first approach of this editor was developed by [PIZA]. As stated in section 3.3.1, this editor is strong coupled to the context, and is also coupled to the Rendering.

During the realization of this thesis, [ZARAGOZA] designed a new Editor with fewer responsibilities than the one by Piza, in order to accomplish with the goal, and finally translating a valid scenario description to lower-level commands. The graphical interface of the Editor developed by Zaragoza is shown in Figure 3-15.

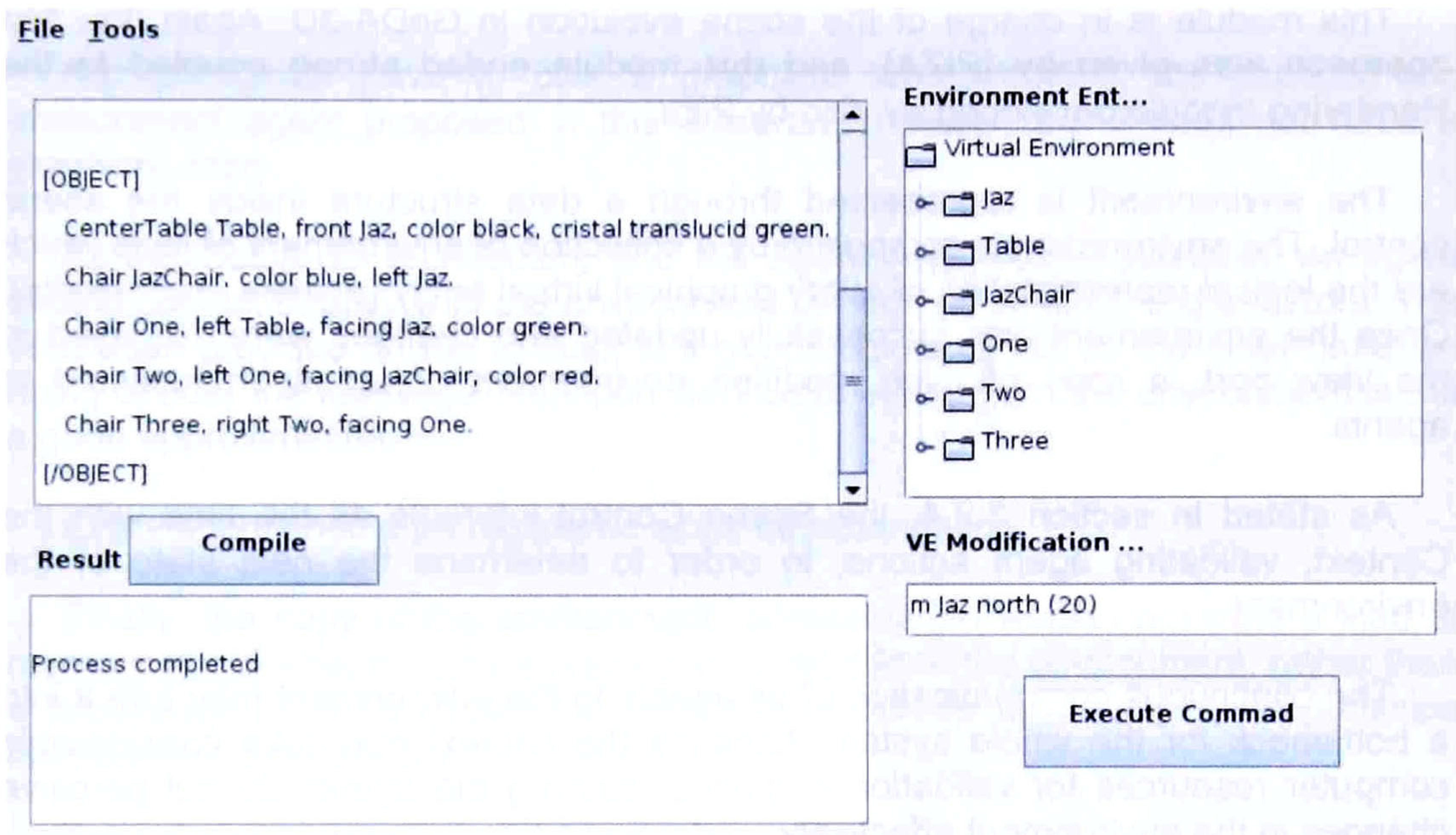


Figure 3-15. Virtual Environment Editor

The scenario description is XML-based (see section 3.2.2.2). This module was developed independently from GeDA-3D and at last was integrated to the project.

This editor only considers the description for the scenario, leaving pending issues related to the scene description. The scene description as stated in section 3.2.2 would be in charge of describing:

1) Personalities of virtual entities and (agent behaviors),
2) Specification of goals to be fulfilled by the entities (the sketch).

3.3.3 Rendering

A first approach of the Rendering was developed by [PIZA]. Once the scenario is displayed in the view port, this module broadcasts to agents the content of the environment. Analogously, whenever the representation of the environment is

changed, the Rendering sends the update to agents. This first module is also strong coupled to the Scene Descriptor also developed by Piza.

A second approach of the Rendering was developed by [MARTINEZ]. This module was developed as a view port, independently from GeDA-3D and, at last was integrated to the project. This Rendering was proven to satisfy its goals through the use of agents which directly communicated with this module.

3.3.4 Scene Control and Environment Representation

This module is in charge of the scene evolution in GeDA-3D. Again, the first approach was given by [PIZA], and this module ended strong coupled to the Rendering module developed by also by Piza.

The environment is represented through a data structure inside the scene control. The environment is composed by a collection of environment objects which are the logical representation of every graphical virtual entity (avatars and objects). Once the environment was successfully updated and changes were displayed in the view port, a copy of each modified *environment object* is broadcasted to agents.

As stated in section 3.2.4, the Scene Control interacts all the time with the Context, validating agent actions, in order to determine the next state of the environment.

The continuous communication of all agents to the environment may turn it into a bottleneck for the whole system, because the context may take considerable computer resources for validation purposes, causing the agents do not perceive changes in the environment effectively

In order to achieve the ultimate goal of this dissertation, the dynamic distribution of the Environment (integrated by the Scenario and the Context), the scene control and the Context should not be inside the Rendering.

3.3.5 Agent Architecture

Analyzing the implementation provided to comply with the agent architecture, the following details were found:

1) The only mean to have an agent is inheriting from class Agent.
2) Class Agent inherits from class DimensionableObject; this means that all agents should have a shape description; that is only visible agents;
3) Class SkillBasedAgent, which inherits from Agent, has a method called performAction() that is used as the effector;

4) The method performAction() receives an instance of class Action; then uses the Message Transport Service to send a description of that action to Scene Control.
5) Every Agent instance must register itself into Message Transport Service to receive messages.
6) The sensor simulation is achieved using the Message Transport Service through a thread that receives an ACLMessage, and then passes such message to agent.
7) Every message incoming from the Rendering module is processed directly by the agent.
8) The agent owns a copy of the environment representation.

Not all agents have to own a graphical representation, for example the environment agent proposed in this dissertation exists, but it does not need a graphical entity.

As the GeDA-3D architecture and the Agent Architecture establish, an agent should contact directly with the environment through its sensors and effectors. The simulation provided for the Effector is a good beginning, but on the other hand, so using directly the Message Transport Service to sense from the environment is not a good approximation.

Every agent should be registered since its instantiation.

Finally, the copy of the environment representation would constitute a kind of memory about what the agent previously sensed from the environment, rather than a knowledge base. A knowledge base would include for example, relationships inferred from the memory.

3.3.6 Agent Platform

Analyzing the operation and implementation of the platform it was found that, once a module is launched on a machine, no other module can use the platform in the same machine if it was not linked to the first module.

It is desirable that we could launch any module at any time from any machine even if the platform is running or not. Of course, it is necessary that the object code of the platform was available to run on the machine. If the platform is not running in local machine, launching a module should start the platform; otherwise the module should link itself to the platform.

The platform does not provide multi-environment support, that is, only one environment can be launched on the platform, because the platform only provides agent ids for delivering a message to its destination.

A White Page Service (a directory of agents) is provided by Message Transport Service through a mechanism to register agents and other services on the platform in order to be localizable. A Yellow Page Service (a directory of services) is not provided.

According to FIPA [FIPA_SPEC] it is normative to provide a Life Cycle Management Service as illustrated in Figure 3-16.

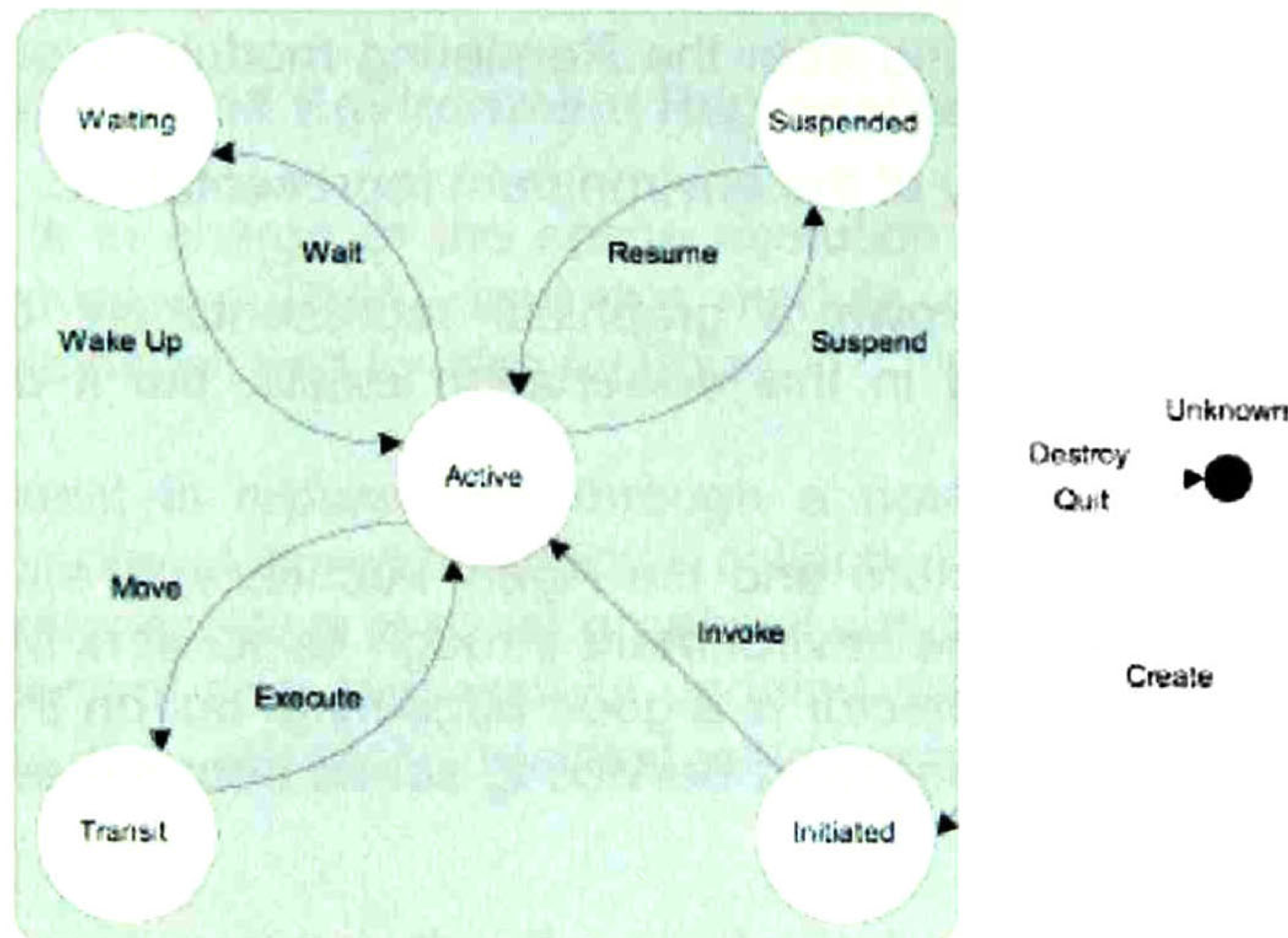


Figure 3-16. FIPA Agent Life Cycle

The platform through Agent Administrator and Core, manages only the states "Unknown", "Initiated" and "Active" with the corresponding transitions; see [FIPA_SPEC] for details.

The Message Transport Service is a key component because all communication between modules and agents depend on it. The following details were found about this service:

1) It uses only ServerSocket and Socket classes, which use TCP (Transmission Control Protocol);
2) It builds a logical mesh topology among all computers;
3) It offers unicast and broadcast but does not offer multicast; so the platform does not support groups of agents
4) Manages individual threads that wait for a message and once it arrives, the thread delivers the message to the addressee (an agent or a module on the platform)

About the use of TCP

A connection-oriented protocol, such as TCP is suitable when we need a reliable protocol that let a flow of bytes originated in a machine is delivered without

errors to another machine. Not all communications in a distributed system need to be treated by as many layers as TCP manages.

Most of communications in a distributed system are in local area networks, which are very reliable. The time spent using a connection-oriented protocol is considerable, especially if the majority of messages transmitted by agents and Render contain less bytes than the payload (65507 bytes) available for UDP (User Datagram Protocol) over IPv4.

UDP is useful for client-server situations and, remembering, the kind of middleware stated in section 3.2.6 is a message oriented middleware. So, it is preferable using the less-connection UDP protocol.

About the lack of multicast

It is necessary the use of groups in a distributed system, especially for the realization of the distributed environment (see chapter 5). Broadcasting involves disturbing nodes not interested in receiving a message, which consumes processing time. Using multicast, a message is delivered only to interested nodes. In the sense of providing an agent platform, it is not a good idea to broadcast a message to all agents when we need to manage groups.

About the logical mesh topology

The building of a mesh in order to connect every computer to each other attempts to system scalability, as the system increases in number of computers and their corresponding LANs. This is the reason why it is not viable continuing extending the current Message Transport Service.

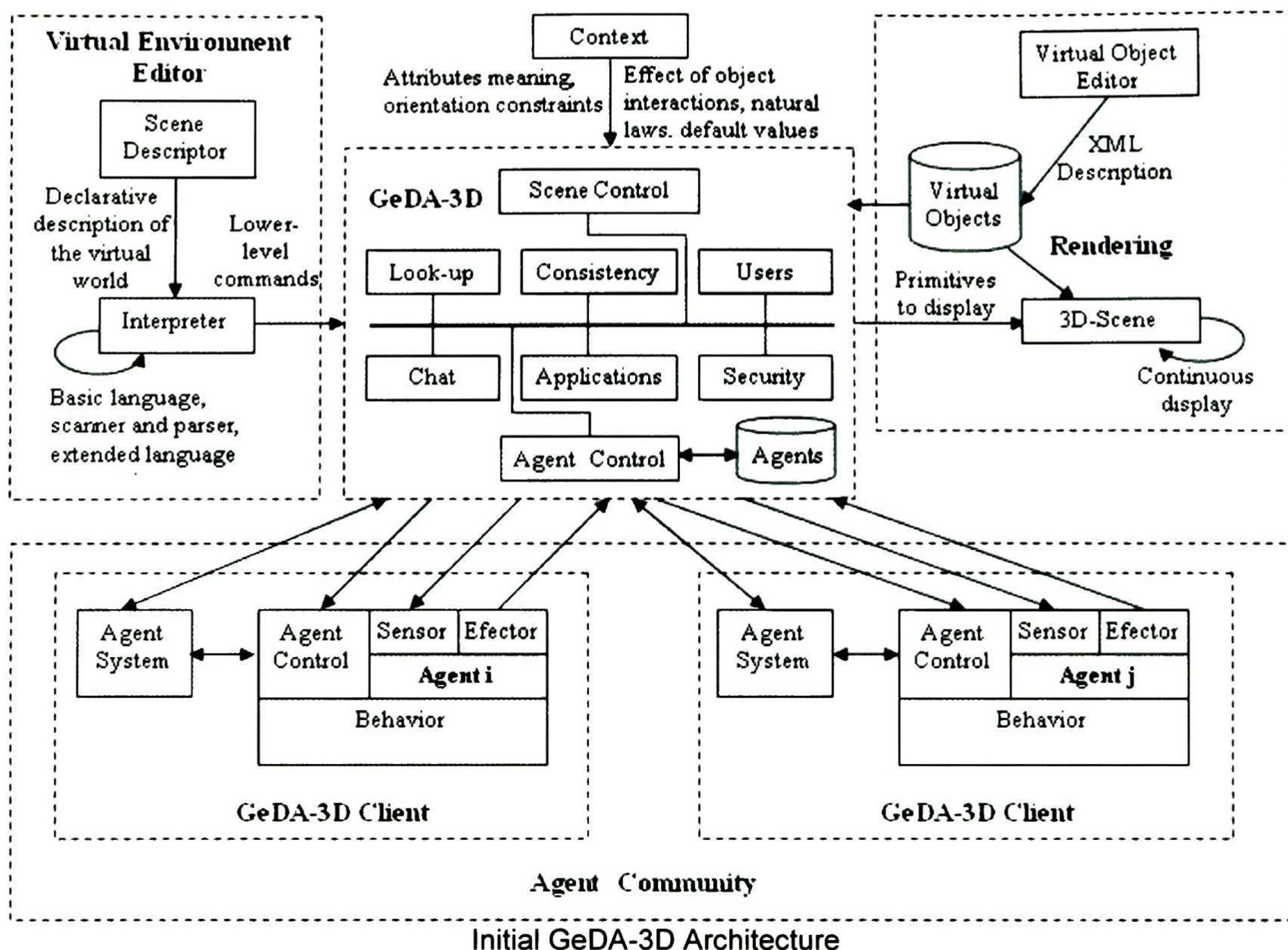
3.3.7 GeDA-3D Architecture

Analyzing the architecture specification illustrated in Figure 3-2 (shown in next page for comfort), and to be consistent with [FIPA_SPEC], the Agent System should be nearer to the platform, next to the kernel, rather than in the Agent Community.

The Agent Control interface mentioned in section 3.2.6.2 should not be only to communicate with the Agent Control; in fact, currently it is used to receive messages from other agents and from the Rendering; so it needs some modification.

The Context module manages many information so there is needed to give sense to the use of a database. This module can be seen as the Ontology service described in [FIPA]. In order to fulfill pending issues related to the scene description, the Virtual Environment Editor should contemplate another component in charge of this.

A good software engineering requires a match between the specification and the implementation. Based on these facts, and the analysis presented in sections 3.3.1 to 3.3.6 and the current requirements of GeDA-3D (see section 3.1), it is necessary a reengineer of the architecture as detailed in section 3.4.



Initial GeDA-3D Architecture

3.4 Reengineering of GeDA-3D

According to the results of the analysis of current GeDA-3D components in the previous section, we propose the following architecture which satisfies current requirements of the project, as illustrated in Figure 3-17.

Following sections detail proposed modifications to GeDA-3D modules.

3.4.1 The Context Module

We propose that the Context module, as described in section 3.2.1, also provide the means to write and read from a database. Such database will contain ontologies, more precisely, descriptions, laws and properties of Avatars, Objects and Worlds.

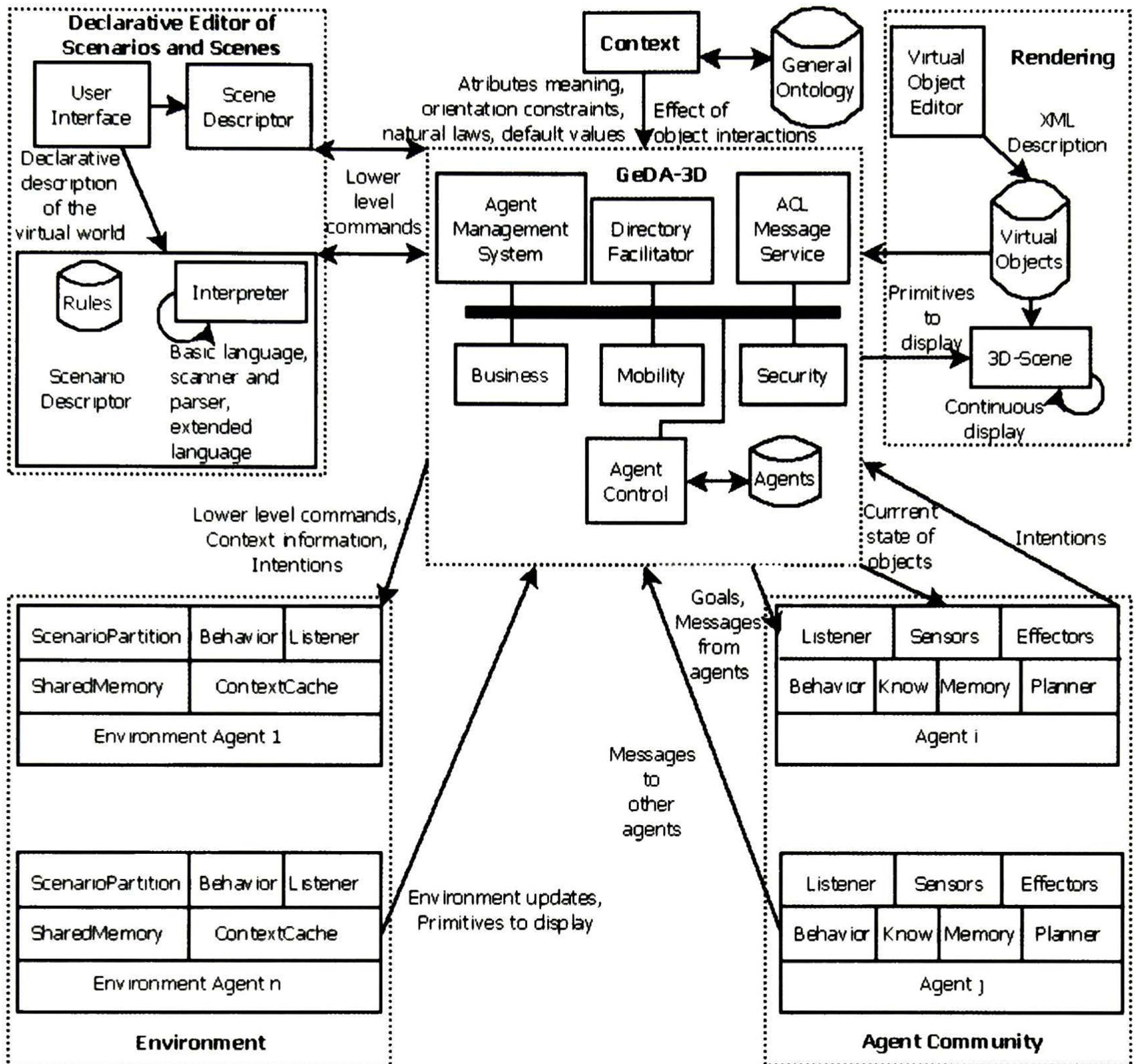


Figure 3-17. New GeDA-3D Architecture

3.4.2 Virtual Environment Editor

We propose a redesigning of the Virtual Environment Editor, containing a User Interface which gives access to a Scenario Descriptor and a Scene Descriptor.

Scenario Descriptor would be represented by the editor developed by [ZARAGOZA] (see section 3.3.2). Scene Descriptor is now just a prototype application which can continue execution only if receives a valid scenario description. Scene Descriptor would be used to describe personalities of virtual entities and specify goals to be fulfilled by the entities (the sketch).

3.4.3 Agent Community

We propose the following updates for the subcomponents of agents.

We propose having a collection of *Sensors* and *Effectors* linkable to agents according to what the end user defines using Scene Descriptor. Such sensors would filter information received from the environment according to the sensor specification. Sensors and effectors are still under construction.

We propose turning *Agent Control Interface* to *Listener*, which remains its responsibility as a listener thread, waiting for incoming messages from any agent or module.

We propose that *Memory* manages a filtered cache of the environment representation which is updated as the entity moves somewhere in the scenario. Such cache would be updated by *Sensors*.

We propose that *Knowledge* represents a knowledge base which would include relationships inferred from the memory.

We propose that *Planner* is in charge of determining the next moves its Avatar should execute in the environment.

About the implementation of Agent Architecture, the class Agent that inherited from DimensionableObject, now is called VirtualAgent. VirtualAgent inherits from Agent. VirtualAgent owns a reference to DimensionableObject and implements an interface corresponding to DimensionableObject methods in the purpose of code compatibility with previously designed applications for GeDA-3D. Also, DimensionableObject must implement its corresponding interface in order to assure that public methods added to DimensionableObject are offered by VirtualAgent.

All classes that inherited from Agent class, requiring shape description capabilities, now inherit from VirtualAgent. Such reengineering is necessary to test the agent platform with previously developed Agents and Rendering proposed by [PIZA] and [ZUÑIGA]. Details about cases of study are exposed in chapter 6.

Now that class Agent was refined we have the following contributions. Given that every Agent should be capable of receiving a message, Agent requires that subclasses implement the IACLMessageListener.messageReceived(ACLMessage) method in order to process an incoming message. This is the relationship established between GeDA-3D and the Listener in Agent illustrated in Figure 3-17.

In section 3.2.5 we stated that an agent is an autonomous process. In order to comply with Agent Life Cycle as [FIPA_SPEC] recommends (see section 3.3.6), it is necessary that class Agent implements an interface that we call RunnableThread. All Agents on the platform should be registered at it. Such interface helps the platform maintain references to all threads in order to send them a signal to stop execution when necessary. Details about process and thread administration are exposed in chapter 4.

The reengineering of class Agent lets now inheriting from it and use all predefined capabilities of agents by all kind of agents. This is very important for the implementation of the Environment Agents that will represent the Distributed Environment (see section 3.4.6).

3.4.4 Agent Platform

Given the scalability problem of the Message Transport Service, we propose its replacement by a *Micro Kernel* which offers the same services, the same interfaces, but using a less-connection protocol, UDP, and of course guaranteeing message delivering. This kernel supports multiple environments running on it. Also, this kernel provides mechanisms to support almost all states recommended in [FIPA_SPEC] for agent life cycle management; the exception is the Transit state. The services provided by the platform are described below.

The *Agent Management System* (AMS) is in charge of registering all agents willing to run on the platform. As [FIPA_SPEC] recommends, AMS provides a white page service, that is, a mean to localize any agent in the platform.

The *Directory Facilitator* (DF) as [FIPA_SPEC] recommends, provides yellow pages services to agents. Agents may register their services with the DF or query for offered services.

The *ACL Message Service* is in charge of facilitating communication primitives to agents. To use the ACL Message Service it is mandatory that the agent was registered with the AMS previously. The tasks this module executes are the same stated for Message Transport Service to send, receive and broadcast messages (either encrypted or not) and include binary content (see section 3.2.6.1):

The *Mobility Service* provides the mechanism to achieve mobility of agents through all machines on the platform. This service permits only *weak mobility* (see section 2.2.5). It is still under construction.

The *Business Service* would provide the means for establishing a negotiation medium, available to agents for generic purposes.

Details about the *Distributed System Platform* design and implementation are exposed in chapter 4.

3.4.5 The Distributed Environment

The ultimate goal of this dissertation is the dynamic distribution of the Environment (integrated by the Scenario and the Context). Based on what was stated in section 3.3.4, it is mandatory to remove the Scene Control from the architecture.

We propose that, Scene Control responsibilities (see section 3.2.4), be handled by Environment Agents. There will be a number of environment agents representing the environment and will control scene evolution. Particular subcomponents of an environment agent are described below.

We propose that the *Scenario Partition* represents just a piece of the whole environment representation (the data structure composed by environment objects, described in section 3.3.4). Such piece must contain all the environment objects needed to process intentions sent by a subset of the agents in the community.

We propose that the *Context Cache* represents a subset of the ontologies managed by the Context Module. Such subset should be a copy of just the rules needed to validate intentions and generate corresponding environment updates. As a cache, it should be updated if necessary when the Ontology changes.

We propose an implementation of a *Shared Memory* with which, an environment agent can determine if it is able to process an agent intention or, such intention must be forwarded to another environment agent. This memory is necessary to obtain information about which agent manages the region of the environment involved with the intention.

Our proposal is based on that an environment agent will be in charge of the scene evolution of a region of the virtual environment. Once an intention is successfully carried out by an environment agent, it sends environment updates to agents and the corresponding primitives to display changes shown by Rendering. Details about the design and implementation of the Distributed Environment are exposed in chapter 5.

3.5 Summary

Re-documentation allows obtaining a compendium of the whole GeDA-3D project, identify key ideas previously established and, understand the reason why it is necessary, the design and implementation of a distributed environment in charge of scene evolution.

Reverse engineering let the analysis of GeDA-3D modules related to scene evolution. This analysis led to the identification if missing mechanisms required to coupling the distributed environment to the GeDA-3D project.

Reengineering of GeDA-3D was made to provide the basis for building the distributed environment. Once GeDA-3D platform was updated, it was easier the implementation of the distributed environment, because the environment agents now have to deal only with policies related to the environment representation and scene evolution. All issues concerning to processes, groups, multi-environment support and communication are delegated to the distributed system platform.

Chapter 4

Distributed System Platform

In this chapter we specify the architecture and system requirements satisfied by our new distributed system platform. This platform must provide services required for agent platforms as recommended by FIPA, and support any distributed application which may run on it, specially a multiagent system which needs to distribute a centralized component like the Environment.

4.1 Introduction

In chapter 3 we established the necessity of redesigning the GeDA-3D platform in order to meet the requirements to support a distributed environment representation.

We decided to use Java for the platform implementation because Java handles transparently issues related to data representation on heterogeneous machines. Also this paradigm allows implementing agent migration easily.

This new platform is constituted by a Microkernel and a set of services required for distributed platforms and agent platforms. As the previous version of the platform (see section 3.2.6), this version also complies with the characteristics of a message-oriented middleware.

The main objective of the proposed platform is to develop 3D applications, based on agent's paradigm. However, this platform is useful to support any distributed application and, with a few extra features it is also used as an agent platform.

All components of this platform are designed to comply with the Model-View-Controller design pattern, have low coupling and high cohesion (see section 2.4.2).

4.2 Architecture

The basic Distributed System Platform architecture designed to support distributed applications and multiagent environments is illustrated in Figure 4-1.

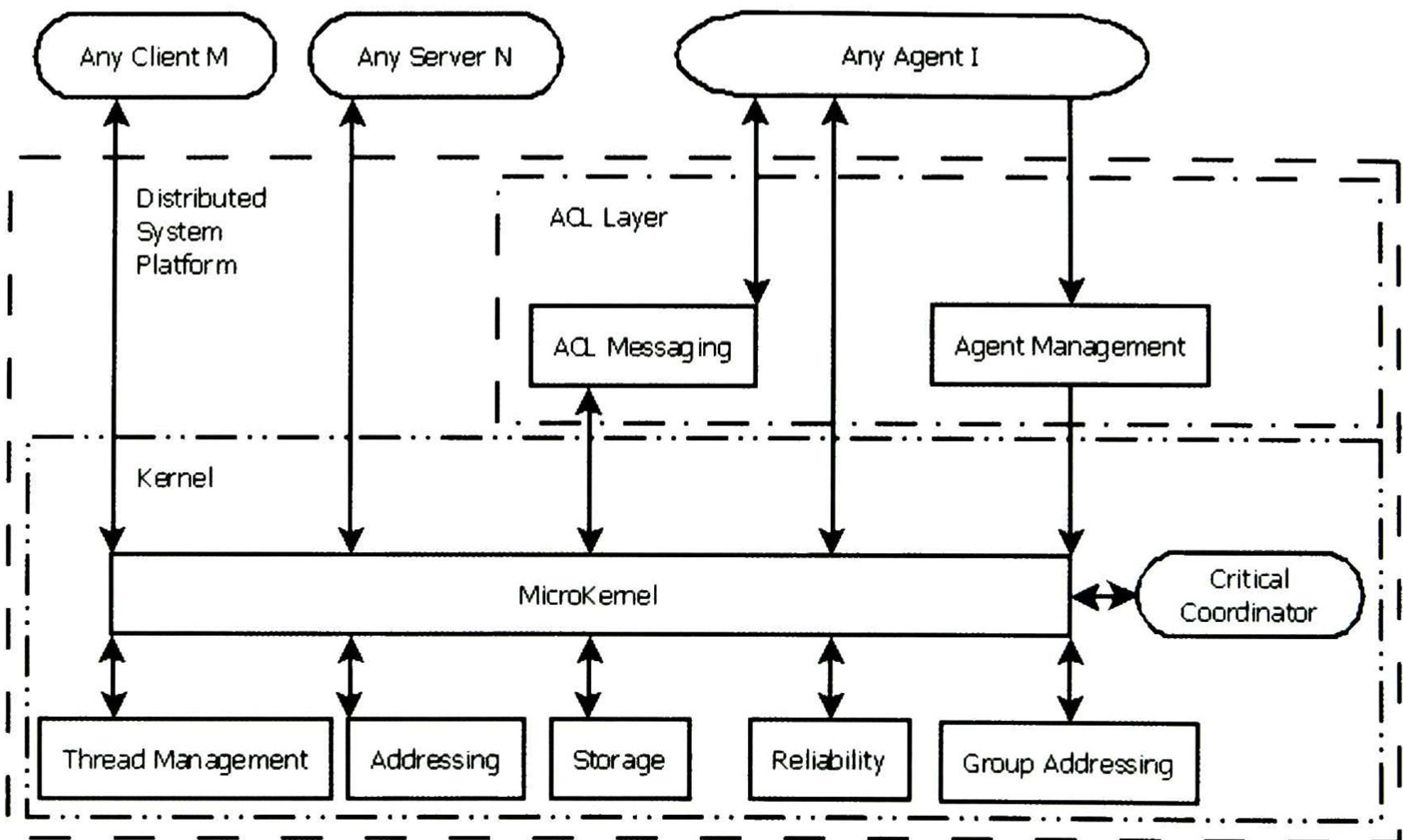


Figure 4-1. Middleware Architecture

The platform is distributed across available computers; each one contains an instance of MicroKernel which in turn contains all shown modules administering information about only local processes. A process requests all type of services through Kernel interface. Services going from to be registered in platform, till, send and receive messages either using communication strategies point-to-point or one-to-many.

The MicroKernel is in charge of sending and receiving all messages and, provides all necessary interfaces to obtain platform services. The Thread Management Module is in charge of registering all local processes and threads in order support all needed management (transparent localization, routing of messages, etc.). The Addressing Module looks up for destination through queering local tables or asking all computers whenever a process (or an agent) needs to communicate with another. The Storage Module is in charge of buffering sent and received messages. The Reliability Module guarantees that any message sent is received by its destination if the latter is traceable. The Group Addressing Module

manages group memberships for processes. The Agent Management Module is in charge of registering all agents and associating them to a process in order to obtain services from the platform. ACL Messaging Module is in charge of sending and receiving ACLMessages, requesting MicroKernel send and receive messages as necessary; it also splits and reintegrates messages if they are larger than the maximum amount of bytes that the platform is set to receive.

4.3 Microkernel

The main component of the platform, Kernel is accessed by all modules in a static manner, that is, there is no need to pass its pointer through method invocations in order to be used.

In a strict sense this Kernel is a microkernel, because, it is only in charge of: a mechanism for process communication, a limited administration of low level processes, basic I/O channels. We continue calling this microkernel as Kernel just by simplicity. In the previous version of the platform, services required one kernel instance for each one. In this version, the Kernel is launched independently of the services; services are linked to the platform dynamically and, two or more services share the same Kernel instance. This Kernel also complies with the four design issues (see section 2.2.1) of the client-server model: process addressing, message-passing primitives either blocking or non-blocking, buffering and reliability for message delivering.

As stated in section 3.4.4, this Kernel replaces the old Message Transport Service, offering more facilities than its predecessor (see section 4.5 for details). In this version, the Kernel acts as a transparent bus to all processes and threads running over it.

Any thread designed to work on the platform must be registered in order to use the platform´s services. The register is done inheriting from class SystemThread. The SystemThread class through its constructor invokes Kernel to request registration of a thread. Upon the system state, the platform may or may not accept the registration request. The system states refusing a registration request are: not launched or shutting down.

Kernel is accessed in a static manner and its instance is managed as a singleton. To launch an application on the platform, the main thread of the application should call the loadClass() method of the Kernel, providing the class name of the application to be loaded (see section 4.5). If the platform was not already initialized on the local machine, then the first application that invokes the Kernel on the local machine causes the platform starting and the Kernel turns available to be used by processes. The View module of Kernel uses a push model (see section 2.4.2) to get the system state, that is, the View module registers by itself into kernel in order to show events occurring in Kernel. However, the Kernel

can run in an invisible way that is, the View of a Kernel can be not available for final users.

Once an instance of Kernel is running on a machine it is not allowed that another application launched another instance of Kernel. The Java singleton mechanism applies only for a java virtual machine (JVM) context, that is, singleton applies only to the JVM process running on the operating system. Another JVM process may be launched on the operating system and could let another application attempt to launch the platform and get another singleton instance of Kernel. In order to have only one Kernel in charge of the services on the local machine, the Kernel launches a thread called LocalPlatformFinder which looks up for an instance of the platform on the local machine. If LocalPlatformFinder finds the platform running on the local machine, In order to allow different application run on the same computer and keep the singleton property, If one application wants to start a Kernel in a computer, first the LocalPlatformFinder is launched, if it doesn´t finds a kernel, then it is allowed to start a new one and continue with following steps. On the other hand, if it finds a running kernel, the request of starting a new kernel is stopped and a process (the idle kernel) sequences the load of classes needed by the second application running on the computer. Once all the classes are loaded, the idle kernel finishes its execution.

The Kernel uses a less-connection protocol, UDP, for message transfer from one machine to other(s). We decided this because most of communications in a distributed system are in local area networks, which are very reliable. The time spent using a connection-oriented protocol is considerable, especially if the majority of messages transmitted between processes, for communication and coordination, contain less bytes than the payload (65507 bytes) available for UDP over IPv4.

The platform uses a set of packets which may be sent by any Kernel when needed and received by one or more kernels depending on the packet. Such packets are treated with the maximum priority and are processed exclusively by a receiver Kernel. These packets are called kernel to kernel (KTK) packets; some of them are described below and more packets are exposed in section 4.5.

OKP (Operating Kernel Presentation): is a presentation packet to announce another machine its kernel availability. If a kernel receives this packet from an unknown machine, then registers the remote machine host address (an Internet Protocol address, IP) and sends back another OKP This packet is necessary for decision making when a kernel requires considering information from others.

OKB (Operating Kernel Bye): announces this kernel unavailability for future decision making. This is sent when a user requests this kernel to shutdown.

IPR (Is Platform Running): asks if the platform is running on the local machine. This packet is sent by the LocalPlatformFinder thread.

PIR (Platform Is Running): it is used to confirm that the platform is running on the local machine as a reply to an IPR packet.

NKL (Notify Kernel Load): notifies the sender kernel's workload for decision making about where to load a process. Every kernel manages a tree data structure to determine the lowest workload machine when necessary.

LLC (Load Local Class): requests to load a class on the local machine where this packet is received. Such class may be a process class or any other. This packet may be sent by an idle kernel instance; if the latter happens then the running kernel may request to load the process on another machine, trying to balance workload.

LCL (Local Class Loaded): confirms that a class was loaded according to the corresponding LLC packet.

To calculate an approximation of the load in a machine we use the following formula criteria. Let l be the load of the machine, c be the amount of cpu used by currently running threads, q be the queue of bytes pending to be read by processes, w be the amount of potentially cpu required by waiting threads if any of them wakes up after receiving work to do, and C, Q, W, A, P and K be constants

$l = (C*c + Q*q + W*w) * A$, where C=0.6, Q=0.3, W=0.1, A=1000000 and
$1 = C + Q + W$, due to the load is expected as a value from 0 to A

$c = 1 - P / p$, where P = 5, P is the NORM_PRIORITY of a Java thread and,

$p = \max(P, \sum_{i=1}^{m} \text{priority}(M_i) + \sum_{j=1}^{u} \text{priority}(U_j))$, where

priority(x) $\in$ {0, 1, ... 10}; priority(x) returns 0 if isWaiting(x) = 1
m is the amount of kernel threads,
u is the amount of user threads,
M_i is a kernel thread,
U_j is a user thread and,

$q = b / A$, where b is the amount of buffered bytes in Mailboxes, pending to be read

$w = t / r$, where

$r = m + u$

$t = \sum_{i=1}^{m} \text{isWaiting}(M_i) + \sum_{j=1}^{u} \text{isWaiting}(U_j)$, where isWaiting(x) $\in$ {0, 1}

The latter formulas are used to differentiate roughly a machine with more user threads running than other or, a machine with higher priority threads running than other or, a machine with more bytes pending to be read than other or, a combination of two or more of these. The values for constants C, Q, W, P and the

values returnable by priority(x) are subject of a future more precise analysis in the matter of load balancing issues.

Once a Kernel is launched, it listens for incoming messages using a reserved port and a dedicated thread with the highest priority; before this thread keeps listening, it launches other seven kernel threads which help it to manage main platform issues:

- MonitorModeMessageHandler is a MessageDeliverer in charge of attending all KTK packets, delegating them to the corresponding module in the kernel (see section 4.5); this thread is also assigned the highest priority;
- UserModeMessageHandler is a MessageDeliverer in charge of redirecting messages to the corresponding MailBox or memory address of a user process; details about message storage and delivering are exposed in section 4.5.3;
- Launcher is always waiting for incoming requests to load classes on the kernel; requests may come from the local machine or a remote machine willing to move its work to this machine;
- RemotePlatformFinder is in charge of sending an OKP packet as soon as possible, and later periodically broadcasting NKL packets to other kernels. In this way, all kernels are aware of the others workload, and anyone can choose the most appropriate machine to load a process; here, we apply an information dissemination technique with a combination of a periodic and event-driven policy (see section 2.2.2).
- ProcessDispatcher is a JobDispatcher in charge of attending processes willing to send a message to a destination; if destination is found then ProcessDispatcher passes the message to NetworkSender; details about process addressing are found in section 4.5.2;
- NetworkSender is a JobDispatcher in charge of sending processes' messages to their destination, keeping a copy of such message and registering departure time; this is also related to message storage-delivering and reliability services exposed in sections 4.5.4 and 4.5.5 below.
- KernelPacketSender is a JobDispatcher in charge of sending and, resending if necessary, KTK packets which are not expected to cause a reply packet message or, the reply packet might be delayed more than 5 seconds. This is in order to assure that a KTK packet is received; this is needed due to the probability of an UDP broadcast packet is not received by destinations. The packages that are requested to be sent by this thread are OKP, OKB, PIR, LLC, LCL (exposed previously) and, RAG and AAG (see section 4.5.7)

A JobDispatcher is a generic thread that queues messages to be processed sequentially. The maximum time that any thread waits before resending a KTK packet is established as 5 seconds as the worst case. The latter value can be set by the system administrator according to an analysis on the target network.

Some threads resend a KTK packet if the corresponding reply packet does not arrive in 5 seconds and, if a reply arrives just after a second request was sent and

such second request does not cause side effects, such request is called idempotent [TANENBAUM]. A side effect can be an inappropriate modification of data or a second launch of a process already launched. The packets sent by KernelPacketSender are treated as not idempotent so, if a kernel receives such packet then registers the arrival of such packet; if a repeated packet arrives, such packet is discarded.

The platform administrator may request Kernel to shutdown, which causes that Kernel sends a terminate signal to all its processes. Kernel waits for all threads to finish their execution but only for a determined time, after which Kernel is no longer available for processes, sends them a finish signal to definitely terminate them, then, the Kernel finishes its execution; shutdown signal applies only to the Kernel in one machine at a time.

4.4 Process and Thread Administration

The SystemThread is the base class for inheritance if an application is willing to run on the platform. An application may contain many threads related to Java class Thread or Java interface Runnable. However, only a registered SystemThread can obtain services from the platform. The relationships between main classes are illustrated in Figure 4-2.

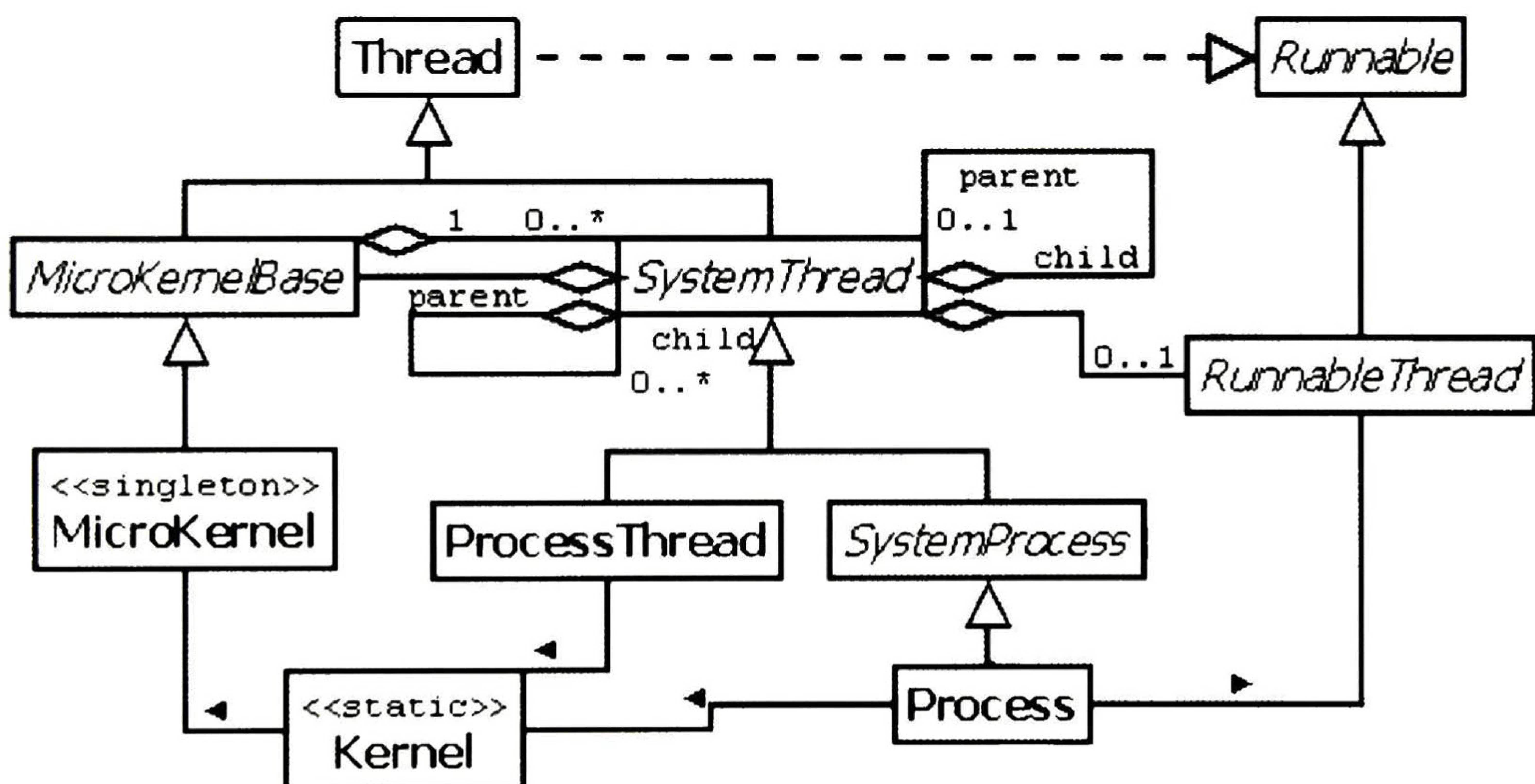


Figure 4-2. Relationship between Kernel and threads

Kernel administrates processes and threads, all threads must belong to a process, this is mandatory, because, the process is the unit of resource assignment. The Class SystemProcess represents a process on the platform. Any process must own a thread and that is achieved through inheritance from class SystemThread.

The constructor of SystemThread is in charge of requesting to register at kernel. Any SystemThread may own children and so its children. The activities to be performed by a SystemThread must be established in the run() method because after invoking the start() method, the JVM will execute that run() method (as with any class that inherits from Thread).

Kernel assigns a unique numeric identifier, a process identifier (PID) of eight bytes length to every process accepted to run on the machine where it is started. Such identifier is unique in the local machine context. The PID is used for message transfer, placing the source PID and the destination PID in every message. Details about message delivering are exposed in section 4.5.3 below.

The Class Kernel provides a set of static methods available by MicroKernelBase and MicroKernel instances in order to provide a subset of public services visible to processes.

The design illustrated in Figure 4-2 shows classes MicroKernelBase, SystemThread and SystemProcess as abstract. MicroKernelBase is the Thread Management module (see section 4.5.1) and the information it handles is necessary for every one of the other modules. The class MicroKernel works as an integrator of all modules (see section 4.5) and kernel threads (see section 4.3) providing interfaces for all platform services. MicroKernelBase is abstract in order to encapsulate its responsibilities, also to avoid its instantiation, and leaving its operations easily available for MicroKernel as if they were part of the MicroKernel source code. MicroKernelBase manages all processes and threads as SystemThread objects. SystemThread is abstract due to the first thread of every process to register at platform must be of class SystemProcess. SystemProcess is abstract due to it must be in the same package as MicroKernelBase and the class to be instantiated (class MicroKernel) is not included in the same package.

Any thread can obtain platform services' from the local machine if such thread has a register PID in the local Kernel and the latter is operational. This means that the Kernel could initialize and a user has not requested the shutdown for the local kernel.

Class SystemProcess requires for its constructors to be provided with the reference of an instance of MicroKernelBase. To do this, the constructors of class Process request Kernel its singleton reference to use it as argument when invoking its parent constructor. This provides an little easier way to instantiate a Process in source code requiring only to import class Process and class Kernel instead of being aware of details like it has to be used the method Kernel.getMicroKernel() and the use of it every time we need to instantiate a process. The same happens with the constructors of class ProcessThread.

Class ProcessThread is the class that represents a thread associated to a Process on the platform. Any Process may create ProcessThreads during the execution of its run() method. Also a ProcessThread may be instantiated granting it

a reference to a SystemThread, which would be its parent. Any request of a ProcessThread to the platform is done on behalf of its process.

When a process or a thread receive a terminate signal, they should terminate all its children. SystemThread provides a method called terminate(), this method is invocated when the Kernel sends a terminate signal to this thread. When terminate() method is invoked, this method invokes the shutdown() method of itself and, after that it requests its kernel to send the terminate signal to its children. The shutdown() method might be overwritten if some activities need to be performed before the thread is unsubscribed from platform.

The inheritance mechanism is not always appropriate for some solutions, for example, a programmer may choose to implement interface Runnable instead of inheriting from class Thread in order program a thread in Java. Analogously, there is available the interface RunnableThread which by the way inherits from Runnable. A reference to a RunnableThread might be provided to constructors of a Process and ProcessThread RunnableThread provides a method named shutdown(). When instantiating a child class of SystemThread, it is possible to provide a reference to a RunnableThread implementer. When the Kernel sends a terminate signal to a SystemThread, and the method shutdown() is invocated, if this method was not overwritten, then it invokes the shutdown() method of the RunnableThread if the latter was provided when instantiating the child of SystemThread.

Every process must belong to a subsystem inside the platform. The PID is composed by two subfields, the subsystem identifier (SID) with four bytes, and the subsystem process identifier (SPID) in that subsystem of four bytes. The subsystem identifier is necessary in order to providing broadcast and multicast only for a subset of processes in such subsystem, without creating and managing a massive multicast group for some needs. Details about process addressing are exposed in section 4.5.2 below.

Processes may be initialized using a process service name (PSN), which is not assumed to be unique for the subsystem. The 32 bit hash value of the PSN character sequence is used as a proposed SPID. In case of various repeated PSNs, it is guaranteed the assignment of a different SPID for each process.

Also processes may be initialized using a subsystem name (SUN), which is treated as unique on the platform. In this case we use a 30 bit hash value from the SUN character sequence due to the first two bits in the PID are reserved as exposed in section 4.5.2 below.

4.5 Platform Services

Kernel provides a set of operations for thread management, process addressing, peer to peer communication, group communication and message

buffering. For every peer to peer communication, Kernel provides a reliable communication. The platform also provides a mechanism to log process events and kernel events in files. All services are provided by their respective modules (see Figure 4.3). MicroKernel provides primary operations to load and run applications on the platform:

loadClass: lets a thread to load a class, in the machine with the lowest workload that the local kernel knows. This operation returns the process unique name of the created process (see section 4.5.2). This method is overloaded; all methods receive the class name of the class to be loaded. Some methods provides the parameter *loadOnAnyMachine* which is used to specify whether no matter which machine is chosen to load the class; the other methods assume a value of true; if the parameter is set as false, then the local machine is used to load the class; the latter is useful when the user wants to load a graphical interface in the local machine. Some methods provide parameters *systemName* and *processName*, which are useful when the user wants to load a process in a determined subsystem and that such process uses a determined PSN in that subsystem (see section 4.4). Some methods provide a parameter called *extraParameter*, which is used for some classes which constructor interface expects three parameters: *systemName*, *processName* and *extraParameter*. Invocations to this operation should be the only instructions of the main() method of an application that a user wants to load on the platform; every java application runs on a different JVM process; the user should request Kernel to load the class of its application, otherwise if the platform is already started on the local machine, all instances of class Process instantiated by the application, will be loaded on another JVM process, the one where the Kernel is operating (see section 4.3) and the application will not be able to communicate with those loaded instances of class Process; thus, instances of Process in the JVM process where that application is running, will not be able to receive platform services because the kernel on that JVM process is idle.

systemExit: requests local kernel to shutdown. This operation may be inhibited previously if a process invokes to addShutdownInhibitor(). Then, only system administrator could request kernel to shutdown.

addShutdownInhibitor: requests local kernel to ignore requests to systemExit() in order to let caller process to perform until finishes.

removeShutdownInhibitor: requests local kernel to unsubscribe its previously invocation to addShutdownInhibitor().

The mobility service will provide the mechanism to achieve mobility of processes through machines on the platform. This service would permit only weak mobility (see section 2.2.5). Because the platform is based on Java virtual machine facilities and the latter does not yet support strong mobility. This service depends on Java serialization and is still under construction.

All modules are allocated at every running kernel so, these modules work in a distributed manner, every one managing only local information and communicating each module with its counterparts when necessary as exposed in following sections.

4.5.1 Thread Management

The Thread Management Module is in charge of registering every launched thread on the platform. As exposed in section 4.4, any process must own a unique PID and this module owns tables that associate PIDs to Java threads and viceversa. When registering a process, this may provide a PSN, so Thread Management Module also owns tables to associate PIDs to PSNs. Given the fact that any process may consist of one or more threads, this module also associates children thread IDs to the corresponding PID in order to attribute any request carried out by a thread to the corresponding process. This module is represented by class MicroKernelBase (see previous Figure 4-2 and the following 4-3). The operations available for users, provided by this module are exposed below:

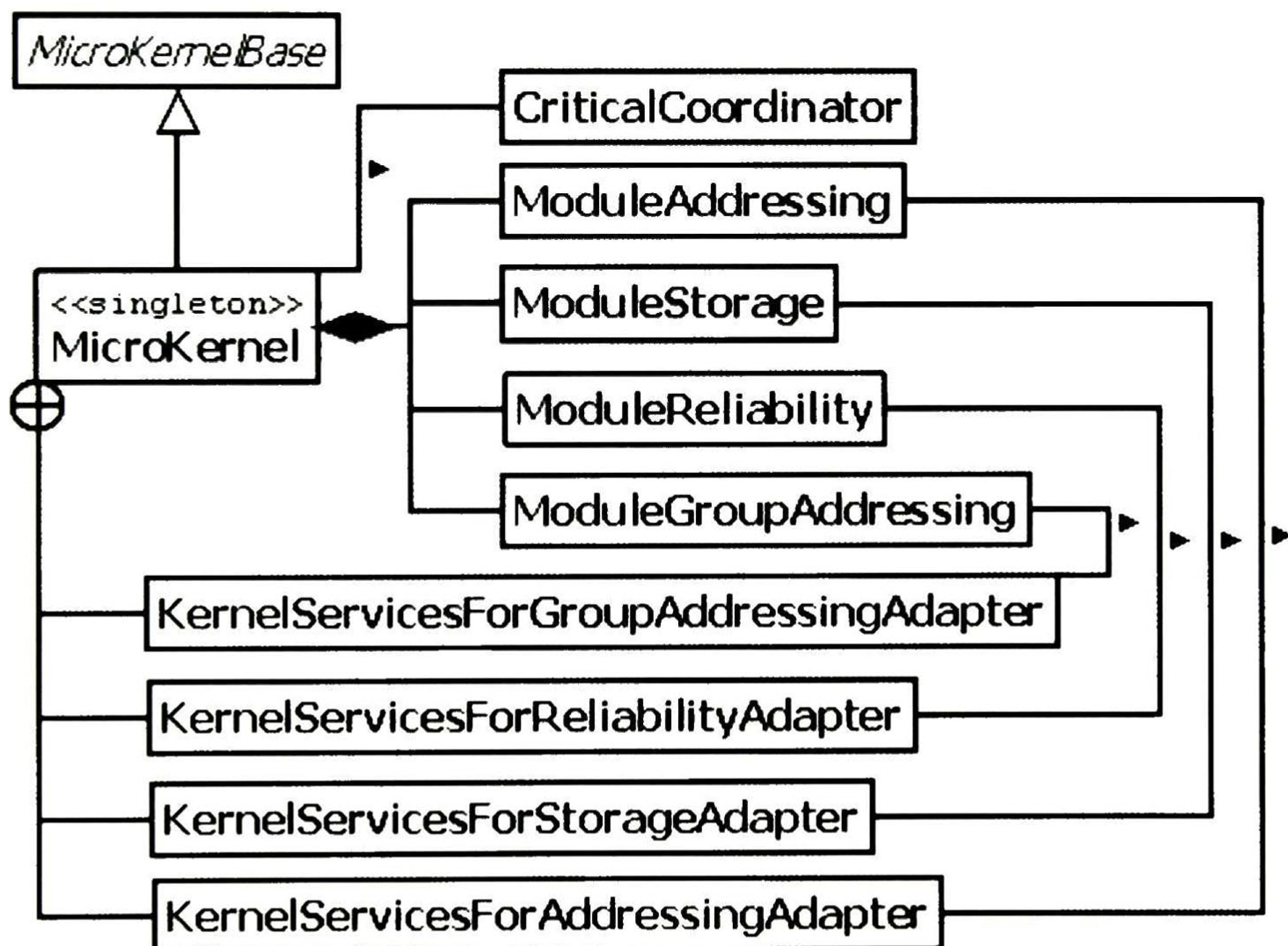


Figure 4-3. Platform Services

suspendThread: lets the current thread to suspend its execution until another thread invokes method resumeThread() or the platform sends a terminate signal to the current thread.

resumeThread: lets an owner of a reference to a thread, to resume the execution of such thread, if the latter has previously invoked to suspendThread(). The owner might be any thread or also the platform.

terminateThread: lets an owner of a reference to a thread, to send the terminate signal to such thread, in order to finish the latter´s execution. The owner might be any thread or also the platform

interruptThread: lets an owner of a reference to a thread, to resume the execution of such thread abnormally, because causes throwing an InterruptedException for such thread. This is useful when a thread has previously invoked to method wait() and there is no other way to resume its execution; for example when the owner requests to terminate that thread. The owner might be any thread or also the platform.

4.5.2 Process Addressing

Every message sent by the platform must contain a source field and a destination field, each of eight bytes long. Such fields are usually used to store PIDs (see section 4.4). Any process in the platform is addressed through the pair machine-process (IP, PID); the IP is the address of the machine where the process with such PID is running.

As exposed in section 4.4, a process may provide a PSN for its registration and, several processes may use the same PSN. The PSN of a destination process is usually used by a sender process when it requests its kernel to send a message to a destination. The PSN is handled analogously as an anycast network addressing because the "nearest" destination process to the sender is the one that will receive the message.

A process might need to communicate with a specific destination process instead of any of the processes using the same PSN. In order to achieve this, each process has a process unique name (PUN), unique in the entire platform. This name is returned by the loadClass() method when a process requests platform to create another process (see beginning of section 4.5). This name can also be obtained by the corresponding process through invocation of the getProcessUniqueName() method. A process *x* reached by a process *y* using the PSN of *x*, may return its PUN in a reply message to process *y*, letting process *y* to communicate with the same process *x* for later messages. The PUN of a process is handled as a unicast addressing because there is only one process using that PUN. A process might request to send a message using either a PSN or a PUN as the destination. The PSN is a prefix in the PUN character sequence, so if using a PUN the destination cannot be found then the corresponding PSN is used.

Based on the options for addressing exposed in section 2.2.1, when a process is willing to send a message to another process, the ProcessDispatcher kernel

thread sends a broadcast of a lookup packet to determine the pair machine-process of the destination. Such packet includes either a PSN or a PUN.

The Addressing Module in each kernel manages local processes registrations using the Addressing Table, a hash table which supports different pair machine-process entries for the same PSN; such table does not distinguish between a PSN and a PUN because it uses a character sequence as the key. The Addressing Module also sends and receives the following KTK packets to achieve the following tasks:

LSA (Lookup Service Address): asks the receiver if it has a registered PSN or PUN. If the latter results affirmative, then the receiver kernel sends back a FSA packet. The LSA packet contains also the PSN and the PID of the requester process. If a kernel sent a FSA, it registers both PSN and PID of the asker for future communications. Only if the local Addressing Table of a kernel does not know the location of the destination process, this packet is broadcasted.

FSA (Found Service Address): confirms that the requested PSN or PUN was found in the machine that received the LSA. It contains the PID and the PSN or PUN of the requested process. The receiver kernel registers the found process, in the local Addressing Table. Many machines may respond to a LSA packet due to the LSA is broadcasted and, many processes of the same kind of service use the same PSN and might be all running at the same time on different machines.

In order to locate processes on the platform, the Addressing Module provides the following operations:

registerAnycastService: registers a process in the local Addressing Table using the PSN. A server process is expected to be sought, thus servers must invoke this method in order to be found. A client process might not will to be sought, if so it does not need to invoke this method.

deregisterAnycastService: requests to unsubscribe the PSN from the local Addressing Table. Any other Addressing Table in a remote machine may keep registration of such PSN. This method is automatically called in the local machine when the corresponding process is sent a terminate signal.

registerUnicastService: registers a process in the local Addressing Table using the PUN. The invocation to this method is optional, if the process is willing to receive messages addressed only to it.

deregisterUnicastService: requests to unsubscribe the PUN from the Addressing Table. Any other Addressing Table in a remote machine may keep registration of such PUN.

getProcessName: returns the PSN provided when registering the invoker process at the platform.

getProcessUniqueName: returns the PUN generated by platform for the invoker process, which could serve to inform another process such name in order to carry out unicast communication between two process. This is possible as long as PUNs are used when invoking send() and sendNonBlocking() methods (see section 4.5.3).

The source and destination fields of a message contain identifiers. Such identifiers might be either PIDs or others. The sender of a message might be the kernel or a process. The receiver of a message might be the kernel, a process, a group of processes, or all processes. The first two bits in the destination field determine the kind of receiver; next table resumes this information:

First two bits	Receiver
0 0	A process
0 1	A group of processes
1 0	Reserved for future use, maybe for kernel processes
1 1	Kernel

Figure 4-4. Bit setting to distinguish kernel packets, process IDs and group IDs

4.5.3 Message Passing Primitives

Based on the options for message-passing primitives exposed in section 2.2.1, Kernel provides the following primitives:

send: requests to send a message to a destination process identified either by a PSN or a PUN. If a PSN is used, it is carried out an anycast communication, so destination may be to a different process of a group in different invocations; using PUN only the specific destination receives the message. This is a blocking primitive, that is, control is returned to the process after the message has been successfully delivered to the destination process or also if destination was not found. It permits sending an array of bytes built by the process. The array should contain all information required by destination process and, formatted as the latter expects.

sendNonBlocking: requests to send a message to a destination process identified either by a PSN or a PUN, where destination is reached the same way as with send(). This is a non blocking primitive because control is returned to the process as soon as the message is copied to the kernel's memory space; the latter is necessary in order to avoid that the message is accidentally corrupted before it is sent, because sender process can instruct to modify contents of the message just after returning from this invocation and kernel could be delayed in sending the queue of messages. It is common to program servers requiring that control is returned as soon as possible in order to attend the next request rapidly and avoid that kernel discards requests (see section 4.5.4). This primitive as send(), also sends an array of bytes built by the sender process.

receive: requests to receive a message. This is a blocking primitive, that is, the control is returned to the process after the message has been copied to the process space. This primitive allows the receiver to receives an array of bytes sent through send() or sendNonBlocking(). This is an overloaded method that in conjunction with a mailbox may obtain the arrival time of the message since it was delivered into mailbox (see section 4.5.4). There is no need to provide a non-blocking primitive because it is a multithreaded platform and some threads can be working while another is blocked waiting to receive.

These primitives work as long as the system is available, that is, the local kernel has been successfully initialized (see section 4.3) and it is not currently finishing. It is important to clarify that these primitives obtain a pointer to an array of bytes as parameter. The first two primitives will send every byte in a single datagram. However Kernel is configured to receive a determined maximum amount of bytes, so clients must be developed considering such amount in order to all data sent is completely received.

4.5.4 Message Storage and Delivering

After a process has invoked to either send() or sendNonBlocking() (see section 4.5.3) the ProcessDispatcher kernel thread requires the Addressing Module to determine the actual location for the destination (see section 4.5.2). When the latter is done, then the NetworkSender kernel thread manages the message. The NetworkSender stores a sequential counter in every message to be sent, requests the Storage Module to keep a copy of such message and notifies the Reliability Module (see section 4.5.5) that a message is about to be sent. Sent messages are kept in Storage Module in case that it is needed to resend any message.

In order to ensure successfully the reception of a message sent by another process, a process requires to previously invoke the receive() method. Given the fact that a process may be occupied processing a previously received message (see section 2.2.1) the platform provides the following operations:

createMailBox: requests to create a MailBox, which is a “long sized” buffer inside the Kernel in order to queue several messages to read in future. The MailBox is fixed sized, so if it is full, does not accept new messages. Messages can be of variable length, so the MailBox stores the size of the message followed by the corresponding content using only the necessary space. This is an overloaded method, if parameter usingArrivalTime is set, then MailBox also stores local time when messages arrive; such time is provided when a process invokes receive() method (see section 4.5.3).

removeMailBox: requests to delete the MailBox created for the caller process. Messages remaining in MailBox are discarded.

When a process invokes receive() method it provides the pointer to a message structure, the Kernel requests the Storage Module to write data in the message. This module either keeps such pointer or if the process owns a MailBox then a message is read and copied to the pointer. If the process does not own a MailBox or MailBox is empty then the invoker thread is blocked.

When a Kernel receives a message, it checks the destination field to search the receiver process on the local machine. If such process is found, then Kernel tries to deliver this message to the Storage Module, otherwise sends a KTK Address Unknown (AU) packet. If this module cannot store that message then Kernel sends a KTK Try Again (TA) packet to the source machine, else Kernel sends a KTK Acknowledgement (ACK) packet. Details about the latter and more KTK packets are exposed in section 4.5.5 below.

4.5.5 Reliability

The Reliability here described concern the sending and receiving of messages. The *Reliability Module* is in charge of keeping track of every sent and received message in order to guarantee message delivering.

The Reliability Module uses the variant request-ACK-reply-ACK (see section 2.2.1). Whenever a message is successfully delivered to Storage Module, in any direction of the communication, the receiver Kernel sends back an ACK packet and establishes in a table of delivered messages that such message was received.

The platform does not try to identify if a message is either a request or a reply, it only keeps track for message delivering, since a message is about to be sent, until such message is stored in the receiver machine either into a MailBox or into the destination process space.

Once the NetworkSender has notified the Reliability Module that a message is about to be sent, this module activates a ChronometerACK. The latter will notify the Reliability Module if it has not been received the corresponding ACK after a determined elapsed time. Some other KTK packets useful during message passing are presented in the next table:

Packet	Meaning	Sender machine	Receiver machine
AU	Address unknown	Indicates that the destination process was not found in local machine	Inhibits ChronometerACK; requests Addressing Module to discard the corresponding pair machine-process; obtains from Storage Module the originally sent message and; requests ProcessDispatcher to send such message using another

Packet	Meaning	Sender machine	Receiver machine
			destination with the same PSN (again as stated in section 4.5.2). If an AU arrived for a PUN, then a PSN is used for the retry.
ACK	Acknowledgement	Indicates that a previously received message has been delivered to Storage Module successfully	Inhibits CronometerACK; requests Storage Module to remove the corresponding message
TA	Try again	Indicates that destination process did not receive a message either because it has not invoked receive() or, its MailBox is full or, after receiving and AYA Kernel determines that it has never received the referred message	Inhibits ChronometerACK; starts a ChronometerTA which after a determined time requests to NetworkSender to send such message; (note that destination will be the same as before)
AYA	Are you alive?	Indicates that for a sent message, an expected ACK has not been received and a determined time expired; the chronometer waits for a determined time to receive an IAA; if no IAA is received it assumes to have received an AU	If the table of delivered messages indicates that such message was received then an IAA is sent, otherwise a TA is sent; if destination is no longer running then an AU is sent
IAA	I am alive	Indicates that the corresponding message was delivered to Storage Module successfully	Assumes to have received an ACK

The basic idea of the behavior of the previous packets can be found at [TANENBAUM], the stated differences were necessary to achieve our proposal goals.

4.5.6 Distributed Mutual Exclusion

In some cases of resource management and decision-making, it is necessary the use of a mutual exclusion mechanism. The platform provides a service for

distributed mutual exclusion implemented based on a centralized algorithm, which consists of managing a server instantiated only in one machine. Such server is called CriticalCoordinator and it queues repeated requests for a list of different resources [TANENBAUM]. This service is used to guarantee group name uniqueness when creating groups (see section 4.5.7 below).

Critical Coordinator is a child class, so its parent class is useful for other resource management necessities when another programmer inherits from that class as illustrated in Figure 4-5. The possibilities attribute shown in Figure 4-5 are special tickets generated to refer a granted resource, so just the owner of such ticket is authorized to use a shared resource.

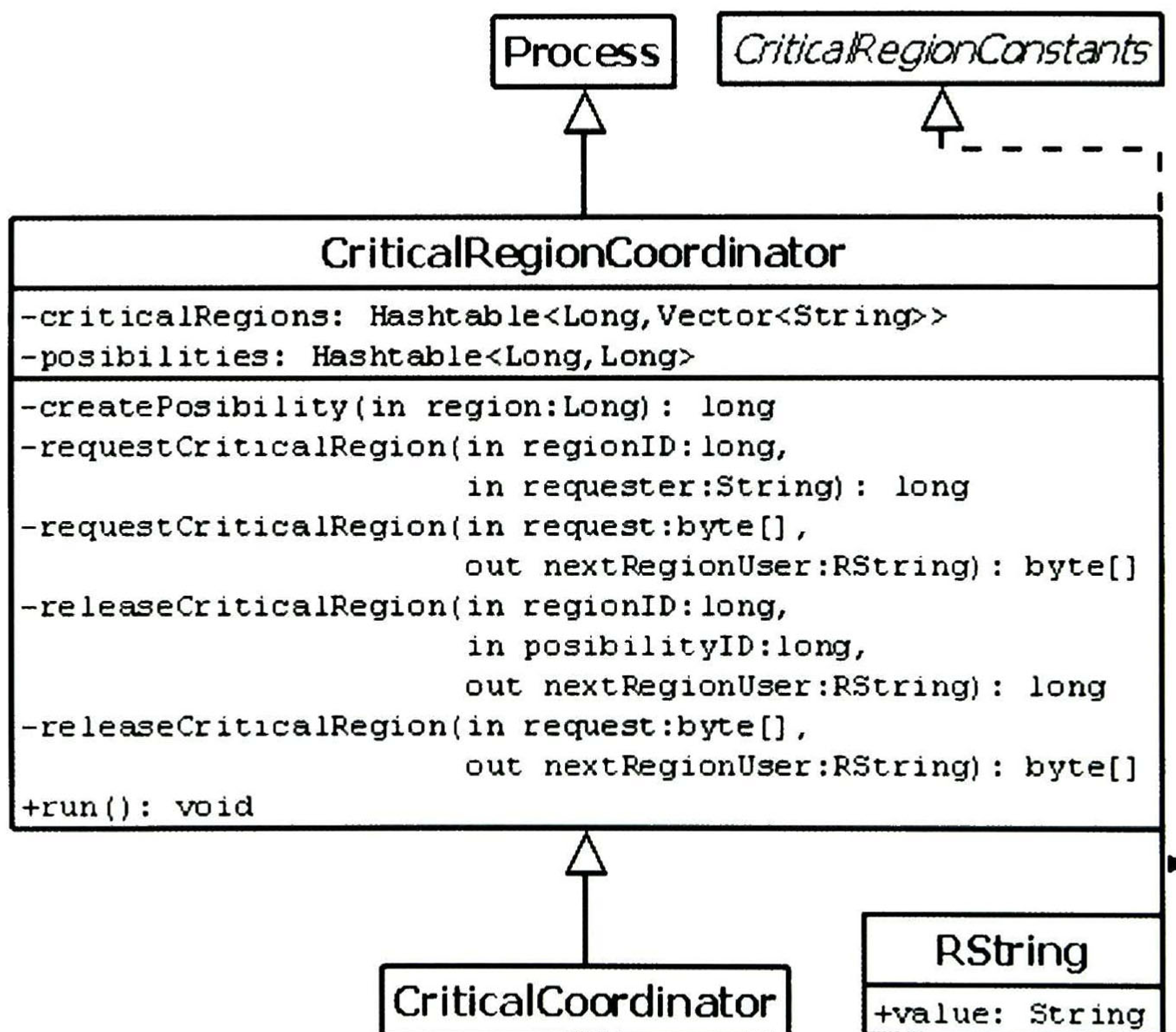


Figure 4-5. The Critical Region Coordinator

4.5.7 Group Communication

The Group Addressing Module is in charge of managing all group issues. For group communication the platform provides the following operations:

joinGroup: requests to join a process *P* to a group. The invoker may be the process *P* or another. The invoker process does not require being a member of such group. The group may or may not exist already; if such group does not exist then it is registered in the platform. If process *P* does not exist, this invocation

does nothing. The process *P* may or may not be part of the same subsystem (see section 4.4) as the invoker.

leaveGroup: requests to unsubscribe a process *P* from a group. The invoker may be the process *P* or another. If the process *P* or the group does not exist, this invocation does nothing. If unsubscribing process *P* from a group leaves the group without members, such group is deregistered from the platform.

sendGroup: request to send a multicast to all members of a group. It is not necessary a receiveGroup() method, so processes receive group messages through receive() method.

sendBroadcast: requests to broadcast a message to all processes in a subsystem. There has not been considered a reason to provide a broadcast for delivering a message to all processes from all subsystems.

The first three of the previous primitives receive a parameter called *group name*, which is a character sequence assumed to be unique. Process addressing uses the source and destination fields to state the counterparts of the communication (see section 4.5.2), so when a group is registered in the platform, it is generated a group identifier (GID) for that *group name*. The GID must be unique and to achieve this when creating a group, Group Addressing Module requests CriticalCoordinator (see section 4.5.6) the resource to be the only module determining a GID all over the platform, because there is an instance of a Group Addressing Module in each kernel; such resource is freed once the GID was determined. To generate a GID or finding the GID for a given *group name*, Group Addressing Module uses the following KTK packets:

LGA (Lookup Group Address): asks through broadcasting all machines whether they have registered a *group name*. If the latter results affirmative then the receiver sends back a FGA, otherwise sends an UGA.

FGA (Found Group Address): informs the corresponding GID for a given *group name*.

UGA (Unknown Group Address): informs that this kernel is not aware of such *group name* existence.

PGA (Propose Group Address): proposes other kernels a random GID to be used, expecting that such GID is not already used for another *group name*. This packet is sent if no FGA packet was received for a given *group name*. The receiver sends back either a RGA or an AGA.

RGA (Reject Group Address): informs that a GID proposed by a PGA is already used for other group that this kernel is aware of its existence.

AGA (Agree Group Address): informs that a given GID is not known by this kernel, so it agrees if such GID is used.

RAG (Remote Action for Group): requests the receiver to be in charge of the rest of the required actions for joining or leaving a group due to this machine has already accomplish its part of the required actions. The receiver sends back an AAG.

AAG (Acknowledge Action for Group): informs the receiver that it was received a RAG in order to free the corresponding invocation for joining or leaving a group.

After a GID for a given *group name* is determined, the PID is associated to the GID for future requests to sendGroup() method.

Group members may be from different subsystems, so a multicast applies for any collection of processes, where they may be members of different subsystems.

Group communication is reliable but is not ordered in time for all members of the group. Broadcast is not reliable at the moment.

4.6 Agent Platform Services

For this platform, an agent is a computational process [FIPA_SPEC] that implements the autonomous, communicating functionality of an application. Agents communicate using an Agent Communication Language (ACL).

All platform services described in previous sections were designed to provide generic services for any distributed application; such services can be used also for an agent platform adding some extra features. An easier way to use the platform for agents is provided through the called ACL Layer, which provides a repertoire of operations for agent management and message transfer. The operations provided by this layer are exposed in sections 4.6.2 and 4.6.3 below.

When registering an agent on the platform, it is required to provide the agent´s name and its corresponding environment´s name. Based on what was established in section 4.4 every process must belong to a subsystem inside the platform. In the same way, an agent must belong to an environment. According to this, we obtain a multi-environment support when registering every agent to an environment. Agents˙ behavior sending unicasts, multicasts or broadcasts in their environment does not affect to other agents and their environments.

An extra benefit of the multi-environment support combined with the group management as they are designed is that, we could launch multiple environments, and one or more agents could join groups from different environments. Thus, an agent could receive messages from two or more environments. As an example we

could imagine an Alpha agent with the role of an engineer into an environment A, which is also a playing a role of a teacher into an environment B; depending on its sensors, effectors and messages received from other agents, this Alpha Agent could behave as necessary in both environments.

For an agent platform in order to be [FIPA] compliant (see section 3.2.6) it is normative that provides:

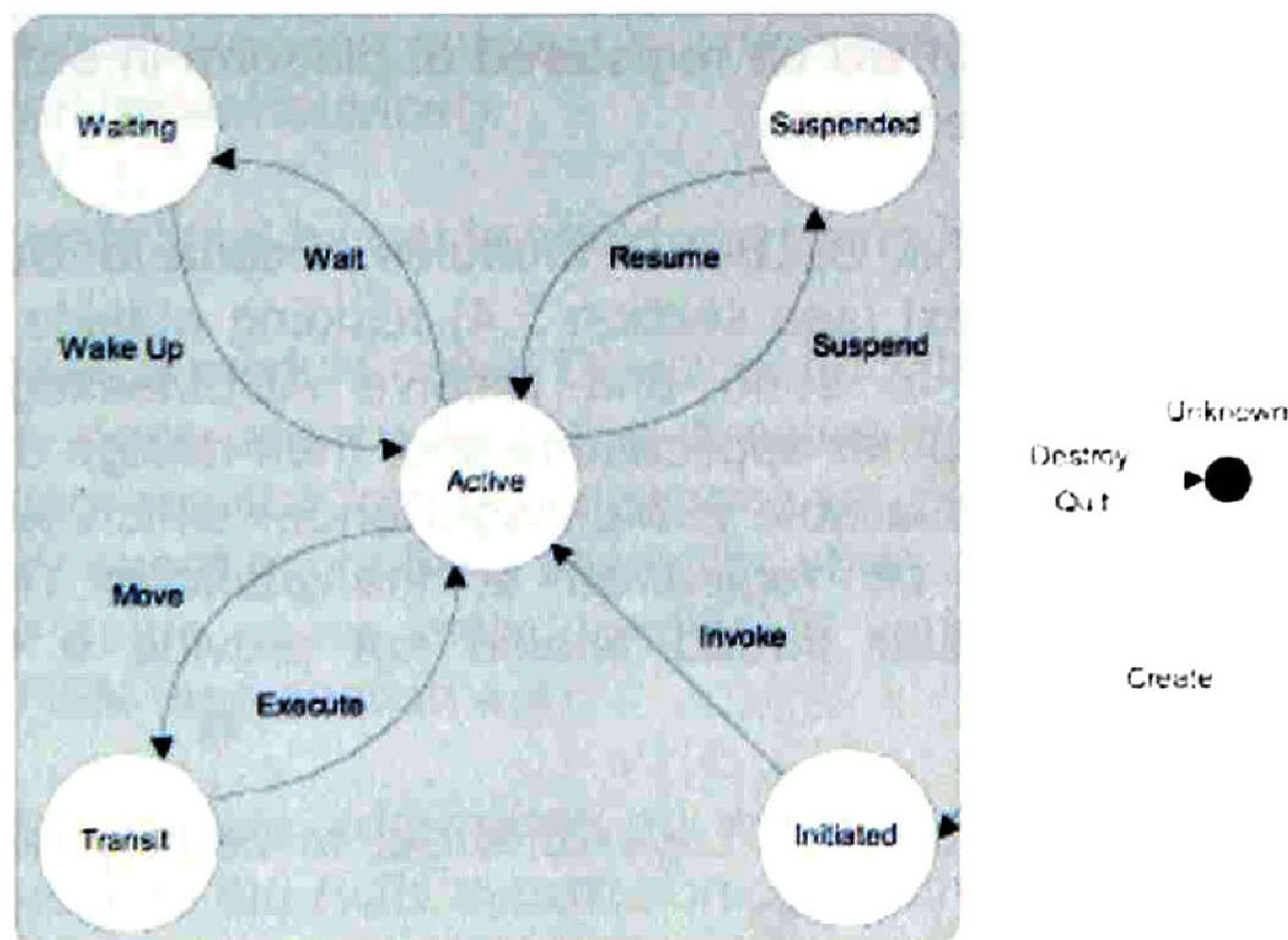
1) Life cycle management
2) White page service
3) Yellow page service
4) Message Transport Service

Details about previously listed services are exposed in following sections.

4.6.1 Life Cycle Management

Most of the agent life-cycle states [FIPA_SPEC] established in Figure 3-16 (which can be seen below for comfort) are make available through the facilities of the Java API for thread management, but the Transit state. The difference between the JVM thread policy and the agent platform thread policy is that, any agent stopped in the platform may be initiated and stopped infinite times. The latter is possible through the creation a new Java thread for the agent when moving from state Unknown to Initiated but using the same instance of such Agent.

The transit state as [FIPA_SPEC] establishes, is not yet supported because this platform does not consider (at this moment) agents mobility between different agent platforms.



The operations that support life cycle management are the following:

Operation owner class	Operation	FIPA state transition
ACL	registerAgent	Create
ACL	deregisterAgent	Quit
SystemThread	finish	Destroy
ACL	startAgent startAgentListener	Invoke
Kernel/ACL	suspendThread	Wait
Kernel/ACL	resumeThread	Wake Up
Kernel/ACL	interruptThread	Resume/Wake Up
Thread	suspend	Suspend
Thread	resume	Resume

Details about previously listed operations are exposed in sections 4.5.1 and 4.6.2.

4.6.2 White and Yellow Page Services

FIPA establishes a module called Agent Management System (AMS) [FIPA_SPEC] which provides a white page service, so every agent must be registered at AMS. AMS handles all agents´ unique IDs and locations. Three modules working together across the platform provide the AMS: the Thread Management Module (see section 4.5.1), the Addressing Module (see section 4.5.2) and the Agent Management Module; the latter is exposed below.

The *Agent Management Module* is in charge of mapping every agent to a process registered at the platform. When an agent requests to register, it must provide a RunnableThread reference (see section 4.4) and a reference to an IACLMessageListener (see section 4.6.3). The RunnableThread reference is used to instantiate a Process which will be registered at platform in order to such agent be able to request platform services.

Some applications like the GeDA-3D modules: Scenario Descriptor, Scene Descriptor, Rendering, Context (see section 3.4) residing outside the kernel, may require using the platform to send and receive ACLMessages in order to communicate with agents or other applications and, their design does not need an agent. In these cases it is available a MessageSender which would be used to instantiate the Process to be registered at the platform. To instantiate a MessageSender, the requester thread should not provide a RunnableThread reference.

The operations provided by ACL Layer in order to comply with a white page service are listed below:

registerAgent: requests to register an agent at the platform. This method receives a pointer to a RunnableThread which run() method would be invoked; also receives a pointer to an IACLMessageListener. The agent´s messageReceived() method would be invoked when a message arrives for such agent.

deregisterAgent: requests to unsubscribe the agent from the platform; this also causes sending the terminate signal to the corresponding agent.

startAgent: requests to start the agent´s thread according to JVM scheduling policies. This method receives a pointer to a RunnableThread which must have been used during registerAgent() invocation. Either this method or startAgentListener() can only be invocated once.

startAgentListener: requests to start the agent´s thread according to JVM scheduling policies. This method receives a pointer to a IACLMessageListener which must have been used during registerAgent() invocation. Either this method or startAgentListener() can only be invocated once.

The following operations work as described in detail in section 4.5 and 4.5.1:

getAgentName: returns the agent´s name using Kernel.getProcessName(). We will refer to this name as the Agent Service Name (ASN).

getAgentUniqueName: returns the agent´s name using Kernel.getProcessUniqueName(). We will refer to this name as the Agent Unique Name (AUN).

loadClass: loads an application using Kernel.loadClass() and returns the AUN of the created agent.

registerAgentUniqueNameForLookup: registers an agent using Kernel.registerUnicastService().

systemExit: requests local kernel to shutdown using Kernel.systemExit().

FIPA establishes a module called Directory Facilitator (DF) [FIPA_SPEC], which provides a yellow page service, this is, a directory of services provided by agents. By default, during registration of an agent (a process) at platform, platform assumes that any agent might be a server for other agents, so we may have more agents using the same name, more specifically various agents can be registered using the same PSN (see section 4.4).

In order to differentiate addressing an agent as a server or as an individual, sender agent must use the AUN to communicate with a specific agent or, using the ASN to communicate with any agent that would provide a service.

4.6.3 Message Transport Service

The agent platform's Message Transport Service established by [FIPA_SPEC] is provided through the called ACL Messaging Service, which is in charge of providing operations for agent communication.

The unit for message transfer is the class ACLMessage which complies with FIPA ACL Message Structure Specification [FIPA_ACL]. ACLMessage provides a repertoire of FIPA compliant performatives and some specific purpose ones. Agents communicate through sending and receiving ACLMessages. Most parameters contained in an ACLMessage are String types. In order to send such parameters through network, they are wrapped into an XML format.

An ACLMessage contains mandatory parameters used for message transfer: sender, receiver, performative and content. Sender and receiver parameters are intended to contain either an ASN or an AUN (see section 4.6.2). Optional parameters are replyTo, language, encoding, ontology, protocol, conversationId, replyWith, inReplyTo, replyBy, binaryContent and arrivalTime. Parameter binaryContent as an array of bytes lets include binary data (like files) in a message. Parameter arrivalTime is used to set the arrival time of message when it was stored in MailBox (see section 4.5.4).

The operations provided by ACL Layer in order to comply with a message transport service are exposed below. The following operations work as described in detail in section 4.5.7:

joinGroup: joins an agent to a group using Kernel.joinGroup().

leaveGroup: unsubscribes an agent from a group using Kernel.leaveGroup().

sendGroup: sends an ACLMessage as a multicast using Kernel.sendGroup().

sendBroadcast: sends an ACLMessage as a broadcast using Kernel.sendBroadcast().

The following operations work as described in detail in section 4.5.3:

sendMessage: sends an ACLMessage using Kernel.send().

sendMessageNonBlocking: sends an ACLMessage using Kernel.sendNonBlocking().

sendEncryptedMessage: sends an encrypted message using ACL.sendMessage().

In order to use Kernel primitives, every message to be sent must be converted to its corresponding array of bytes representation. To achieve this it is invoked

ACLMessage.toXML() method which returns an XML String wrapping all parameters, after this such String is converted to its array of bytes representation.

Besides requesting Kernel to send a message, ACL Messaging Service manages messages analogously to OSI Transport Layer. Considering that a kernel is prepared to receive only a determined maximum amount of bytes per datagram (see section 4.5.3), a message to be sent can be partitioned into fixed size packets when necessary.

When registering an agent at platform, after creating its corresponding process, it is also created a thread called AgentListener using the provided IACLMessageListener reference during the registerAgent() method invocation. IACLMessageListener is an interface that may be implemented by any class. The implementer will receive every message that AgentListener receives.

AgentListener is a child thread of the Agent Process, in charge of requesting Kernel creation of a MailBox (see section 4.5.4), after, this listener invokes Kernel.receive() and for every message received it verifies if the latter is a packet of a message. In case of receiving a packet instead of a complete message, listener creates a structure of the expected size for the full message and then copies every packet received in their corresponding position of such structure. Once all packets of a message have arrived, listener creates an ACLMessage instance containing all parameters, binary content and optionally its arrival time if such data was requested by the Agent when registering at platform

4.7 Summary

Kernel requires a mandatory registration of any thread that will request a service to the platform. Thus, only registered threads are served by the platform.

Kernel uses UDP for message transfer due to that a connection-oriented protocol is excessive to handle most coordination messages like the Kernel to Kernel (KTK) packets (see section 4.3) and also most of messages sent by processes and agents. Every message sent and received must be an array of bytes.

The set of services that Kernel provides are organized in various modules: Thread Management, Addressing, Storage, Reliability, Critical Coordinator, Group Addressing and MicroKernel; these modules are accessed through interfaces provided by class Kernel.

MicroKernel is in charge of sending and receiving all messages and, provides all necessary interfaces to obtain platform services. Thread Management Module is in charge of registering all local processes and threads in order to be localizable and manageable. Addressing Module looks up for destination through queering

local tables or asking all computers whenever a process (or an agent) needs to communicate with another. Storage Module is in charge of buffering sent and received messages. Reliability Module guarantees that any message sent is received by destination if the latter is findable. Group Addressing Module manages group memberships for processes. MicroKernel is helped by various kernel threads in order to process some KTK packets, other KTK packets are forwarded to their respective modules.

In order to provide services as an agent platform there are modules Agent Management and ACL Messaging; these modules are accessed through interfaces provided by class ACL. ACLMessages must be transformed to an array of bytes representation to use Kernel operations for send and receive messages.

Agent Management Module is in charge of registering all agents and associating them to a process in order to obtain services from the platform. ACL Messaging Module is in charge of sending and receiving ACLMessages, requesting MicroKernel send and receive messages as necessary; it also splits and reintegrates messages if they are larger than the maximum amount of bytes that the platform is set to receive.

Chapter 5

Adaptive Distributed Multiagent Environment

In this chapter we describe our proposal to control workload for the *n* agents that will represent the distributed environment for each time *t*. This proposal applies techniques of distributed shared memory and load sharing, to distribute and locate information while executing dynamic adaptation policies at runtime.

5.1 Introduction

Virtual Reality (VR) is represented by different interacting entities allocated into an environment, and such entities are designed to carry out some task.

Environment in VR is the scenario where a scene takes place. In our implementation the *Environment* is the entity that contains information of the environment's state and may also contain rules about how to modify it. Context, in our approach, is the entity that contains information about the Environment where an agent community activity is developed. For example, the context in the real world contains physical laws that rule our world, semantic concepts of words, relationships between existent entities in the world, etc. Thus the Scenario and the Context together represents the Environment.

During scene evolution, agents evolve according to their goals and perceptions about the environment, objects and other agents around them. This means that each agent consults the environment to perceive from it, and also to notify it any intentions to carry out modifications to its state, as well as to validate in the context the rules for permitted changes.

Administration of a dynamic 3D environment is difficult because the continuous communication amid all agents and the environment. This communication may turn the environment into a bottleneck for the whole system. This, because the context inside the environment may take considerable computer resources for validation purposes, causing the agents do not perceive changes in the environment effectively.

The GeDA-3D's Environment currently implants two of its modules, the Scenario (MS) and the Context (MC) in a centralized manner.

To alleviate the bottleneck problem, we propose to distribute the scenario and the context adaptively into a set of agents.

We propose an administration of the environment based on special autonomous agents denominated Environment Agents.

The final goal is to distribute both modules through available processors (or machines) over the network, so an entity (an agent) represents one part of the Environment and the whole environment be represented by all entity agents as illustrated in Figure 5-1.

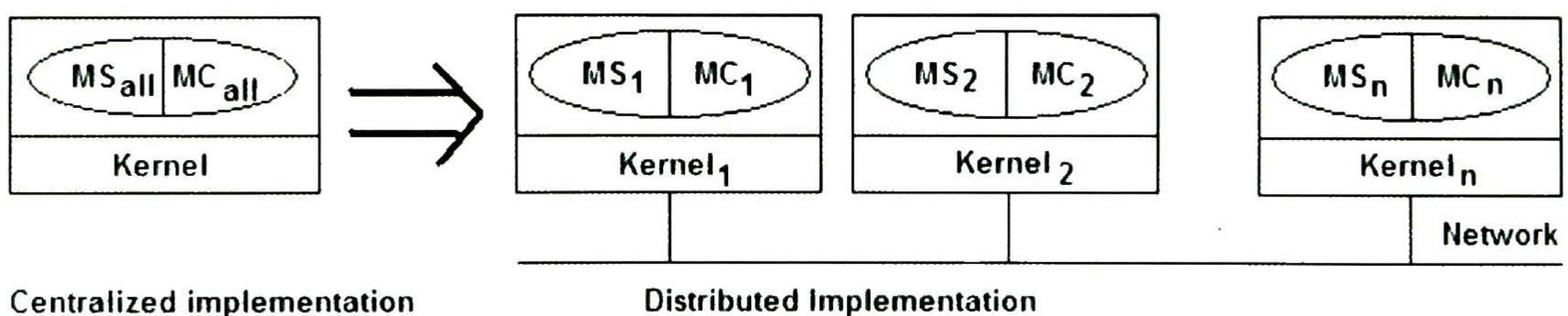


Figure 5-1. Distributing modules Scenario and Context

The distribution proposed must assign (computer) resources in function of the complexity of the scenario at runtime. As the environment changes as time goes on, such resource assignment must be done dynamically. That is, when the complexity grows, more agents must represent the Environment, vice versa, when the complexity of the environment is reduced, the number of agents must decrease.

5.2 Architecture

The design principle we propose for the GeDA-3D environment distribution is based on Environment Agents representing both modules Scenario (*MS*) and Context (*MC*). An Environment Agent (EA) is in charge of administering a region *r* consisting of a MS partition and a MC cache as illustrated in Figure 5-2.

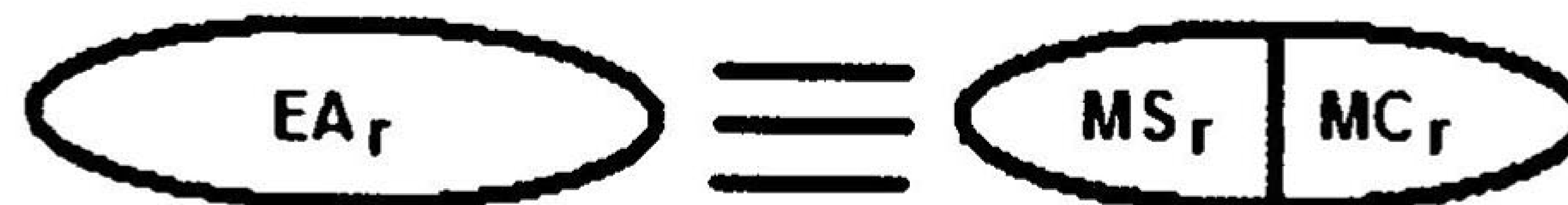


Figure 5-2. Environment Agent in charge of region *r* and its modules

Let *x*, *y* and *z* be integers, the 3D environment is composed by a set of *x*y*z* cubical spaces called *cubes*; for example in Figure 5-3, *x*=3, *y*=3 and *z*=3. The position of each cube is given by the triplet (*y*, *x*, *z*). Assuming that yellow cube is at the front of the scene then it is located at (1, 1, 3) and light green cube is at (3, 3, 3).

A region *r* is an exclusive subset of one or more *cubes*, which may or may not stand next to the others (see section 4.3).

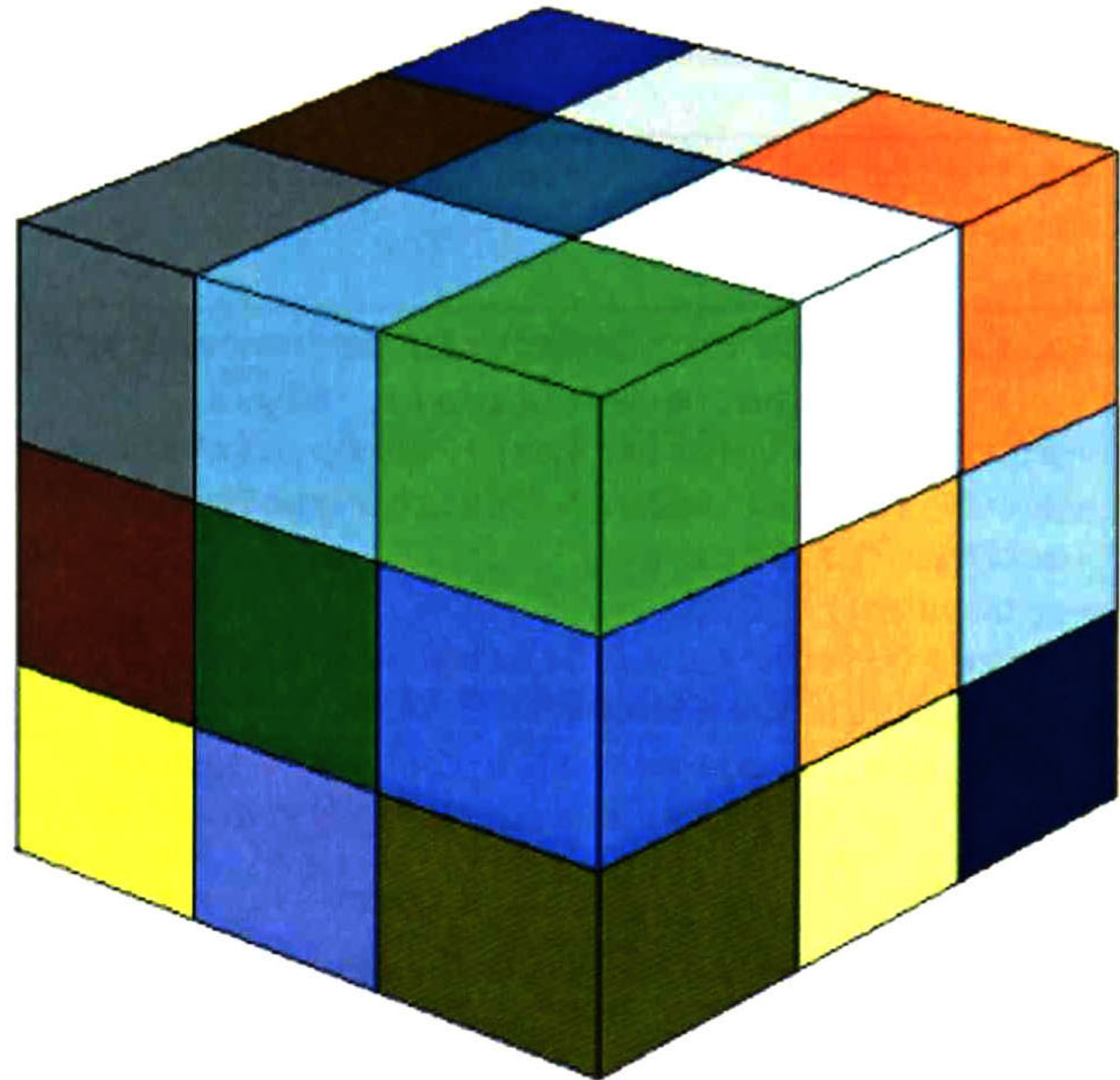


Figure 5-3. The Environment divided into 27 cubes

We propose that the Scenario Module contains a collection of Environment Objects, a cubical space description and the environment´s name as illustrated in Figure 5-4.

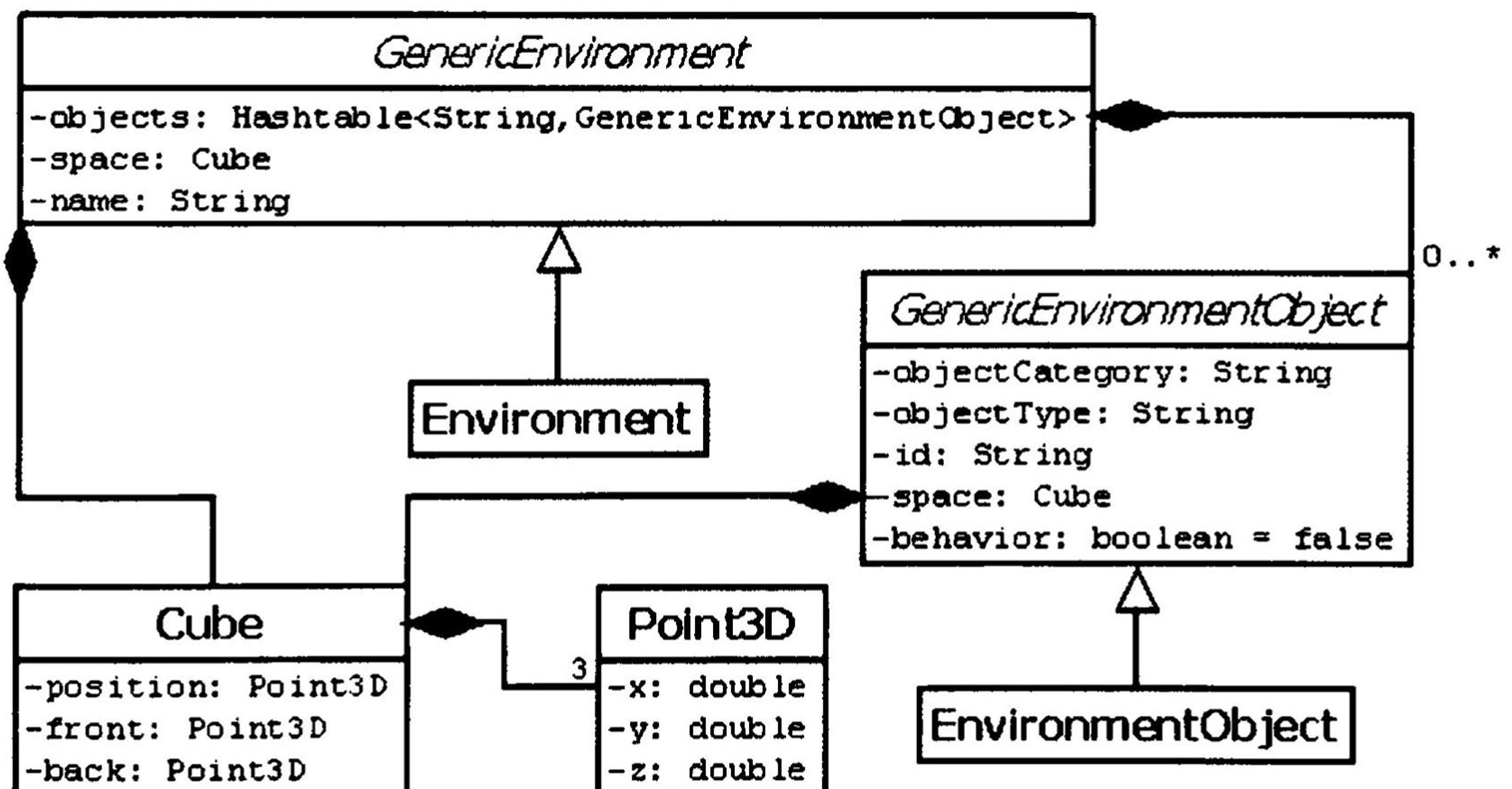


Figure 5-4. The Scenario Module

We propose a class GenericEnvironment that represents a 3D environment. This class owns the environment´s name, is composed by a collection of GenericEnvironmentObjects and, provides basic operations to manage any designable environment as illustrated in Figure 5-5.

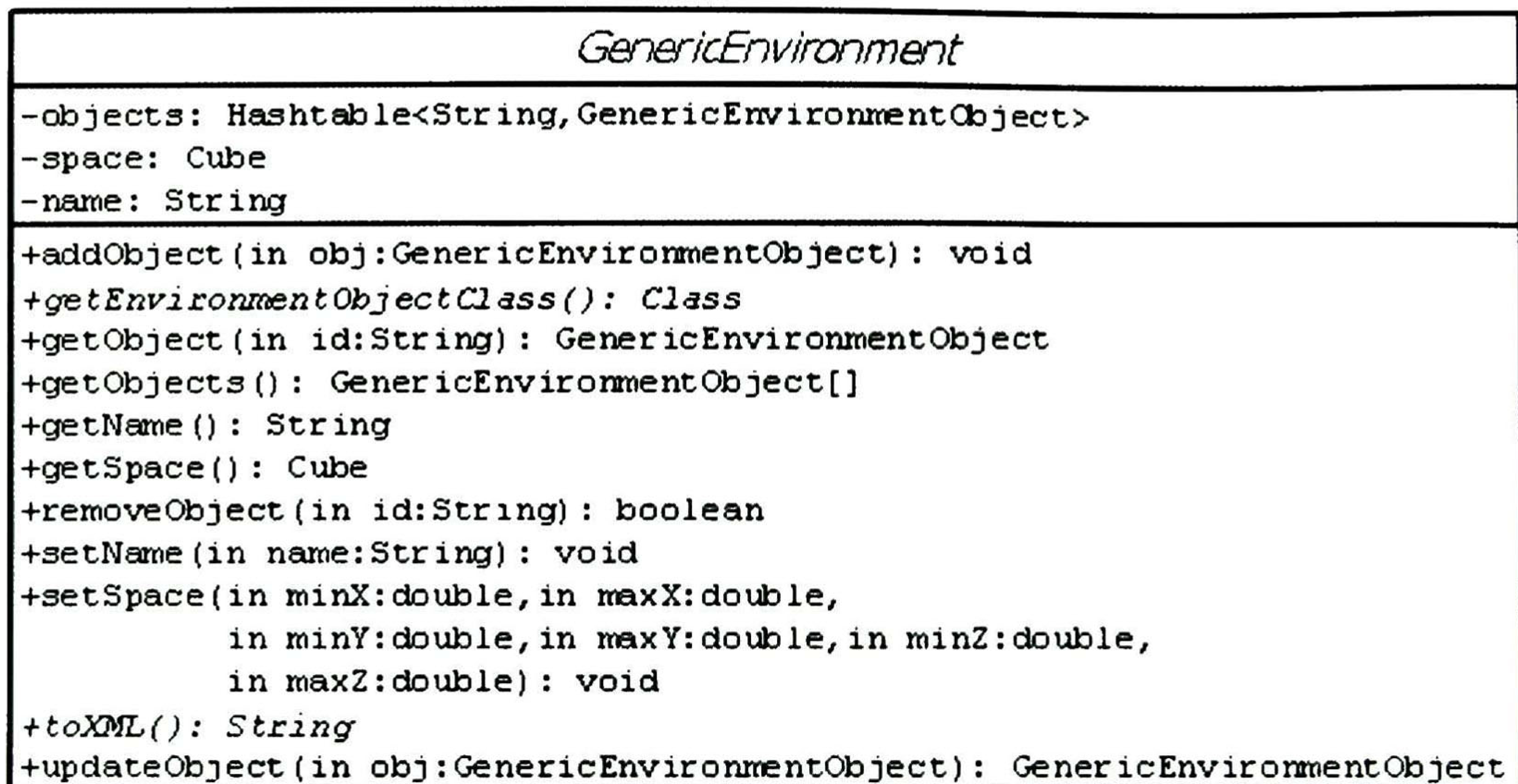


Figure 5-5. The Generic Environment

As seen in Figure 5-5, class GenericEnvironment is abstract in order to let designing of a class Environment, stored in some package and, in charge of specifying the details about the management of the target environment that will be represented. Class GenericEnvironment provides two abstract methods that must be implemented in children classes. The getEnvironmentObjectClass() method is expected to return the specific EnvironmentObject class type that will be managed by the target Environment; this method is necessary because GenericEnvironment manages a collection of GenericEnvironmentObjects (see Figure 5-4) and, when the Environment Agent receives the scenario description (see section 3.4.6), EA loads each EnvironmentObject according to the specified Environment. The toXML() method is expected to return an XML-based description of the scenario and such description is dependent on the specific Environment implemented and the Rendering Module being used (see section 3.2.3).

We propose a class GenericEnvironmentObject (GEO) that represents an environment object in a generic environment as illustrated in Figure 5-6. We propose that a GEO specifies an *object category* as one of either an *Avatar* or an *Object*; where an Object would be a graphical virtual entity and an Avatar would be an Object with behavior. Also a GEO specifies an *object type*, in order to establish a different name for each type of Object or Avatar as they should look different as graphical virtual entities. Every GEO should own a unique *id* in order to be distinguished in the environment due to we may will to show two or more Objects or Avatars of the same *object type*. To ease testing during simulation we use a cubical *space* as the surrounding space of each environment object. Finally, due to

object categories may be different in future works a GEO specifies if such object has a *behavior* in order to provide information about whether this object represents an agent in the environment.

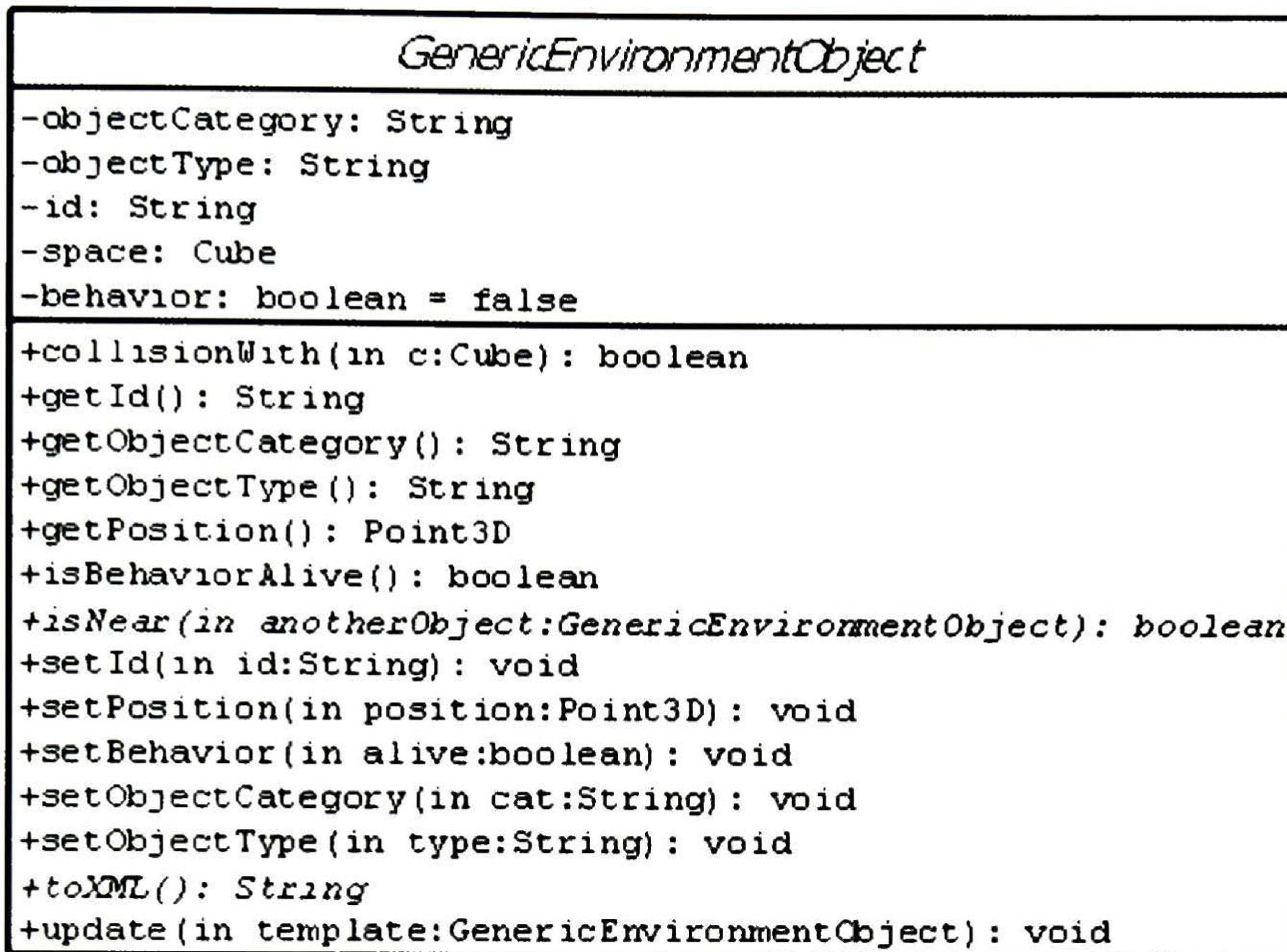


Figure 5-6. The Generic Environment Object

As seen in Figure 5-6 class GenericEnvironmentObject is abstract in order to also let designing of a class EnvironmentObject, stored maybe in the same package that class Environment and, in charge of specifying the details about the management of the target environment object that will be represented. Class GenericEnvironmentObject provides also two abstract methods that must be implemented in children classes. The isNear() method is expected to return whether the current environment object is near to another environment object; this is useful to determine the proximity of two objects according to the implementation of sensors and effectors (see section 3.4.3). The toXML() method is expected to return an XML-based description of the object and such description is dependent on the specific Environment implemented and the Rendering Module being used (see section 3.2.3).

We propose a class Cube as illustrated in Figure 5-7. A Cube determines the bounds of a component of the graphical environment (see previous Figure 5-3). A Cube specifies the *position* of the component and its *front* and *back* bounds; where *front* is expected to set the most positive values for coordinates *x*, *y* and *z* of the component and, *back* is expected to set the most negative values. A Cube is also useful to specify the bounds of the entire graphical environment and, to specify the *position* of a graphical virtual entity inside the virtual environment and its *front* and *back* bounds (see previous Figure 5-4).

Cube
-position: Point3D
-front: Point3D
-back: Point3D
+collisionWith(in c:Cube): boolean
+contains(in point:Point3D): boolean
+getPosition(): Point3D
+getFront(): Point3D
+getBack(): Point3D
3
Point3D
-x: double
-y: double
-z: double

Figure 5-7. The Cube

We propose a class EnvironmentAgent as illustrated in Figure 5-8. The EnvironmentAgent is an Agent in charge of administering the Scenario of the virtual environment and also in charge of validating all permitted changes to the environment according to rules established in ContextCache.

Agent
AgentListener
GenericEnvironment
Environment
EnvironmentAgent
-environmentClassName: String
-groupers: HashSet<AgentGrouper>
-adaptationInProgress: boolean = true
-agentsCreatedByThis: int = 0
GenericEnvironmentObject
EnvironmentObject
AgentGrouper
Splitter
Fuser
ContextCache
EnvironmentAgentFrame
0..1
MembershipTable
DistributionMap
Cube
3
Point3D
0..*
0..*
1..*
EnvironmentCube
-members: HashSet<GenericEnvironmentObject>
-neighbors: HashSet<GenericEnvironmentObject>
-administrator: String

Figure 5-8. The Environment Agent

The EA requires to be provided of the *environment class name* of the corresponding class Environment that is going to be loaded; such data is provided by the Virtual Environment Editor (see section 5.3).

The Context Module (see section 3.4.1) may store a huge amount of information about rules for permitted changes to every object in the environment according to each object properties and effectors. Such information is subject of previous validations before it is published to be used by the environment.

Paying attention to the grain size of computations [TANENBAUM], if an EA requested to the Context Module to validate each intention, this would involve overhead of message passing through the platform, exhibiting *fine-grained* parallelism. Given that not all types of objects are managed for a scene and therefore, only a subset of the information of the Context Module is necessary and even more, an EA will only administer one part of the entire environment. Thus, we propose the EA owns a cached copy of such information (see section 2.2.3) in order to exhibit *coarse-grained* parallelism. For every agent intention the EA receives, the latter validates permitted changes in ContextCache. We chose a cache instead of a replicated copy because there is no need that the EA provided means to modify the Context information, so that information is used as read-only.

The EA is helped by *groupers* which are inner class ProcessThreads (see section 4.4) in charge of requesting to subscribe or unsubscribe agents from *agent groups* (see section 4.6.3) whenever the Scenario changes. Every *agent group* created is registered in the MembershipTable. An *agent group* would be created for every object that could be perceived through agent´s sensors or was reachable through agent´s effectors. In this proposal we use the EnvironmentObject´s isNear() method to simulate sensors and effectors and such method tests whether the Cubes of two EnvironmentObjects overlap.

The EA is dedicated to processing agent intentions until it reaches either an upper or lower bound of complexity, after which it will carry out either a dichotomy process or a fusion process respectively. The Splitter is an inner class ProcessThread in charge of cooperating with the EA on the dichotomy process. The Fuser is an inner class ProcessThread in charge of working on the fusion process. The flag *adaptationInProgress* is used for mutual exclusion among threads accesing concurrently to the DistributionMap. The dichotomy and fusion processes are exposed in section 5.4. The Environment Agent´s responsibilities and the information managed by the DistributionMap are exposed in section 5.3.

5.3 Distributed Virtual Environment

When the scene and scenario descriptions are ready in the Virtual Environment Editor (see sections 3.2.2.1 and 3.4.2) it will request platform to load an Environment Agent. The Virtual Environment Editor (VEE) will provide to the

Environment Agent the environment's name and specific *environment class name* that will be used to represent the environment. Once the Environment Agent is running, the VEE will send to it the environment description. At first, the entire environment is represented and administered by just one Environment Agent; this and any EA will be in charge of:

- receiving any agent intentions to modify the environment's state,
- validating applicable changes into the ContextCache,
- knowing each entity relationship with others and,
- propagating all permitted changes only to agents interested in such changes

All these activities increase workload of the Environment Agent.

In order to distribute load, an Environment Agent may decide to apply a dichotomy to it, by means of cloning. It divides its information and workload with its clone agent. Vice versa, if it finds itself almost or already idle, it may decide to fuse with another Environment Agent. This behavior is achieved in function of the complexity of the 3D environment.

The proposed solution in this work consists in distributing the environment adaptively on a set of Environment Agents as illustrated in Figure 5-9.

Each Environment Agent represents a volume *x* of the environment and, all of them are perceived as a single entity by other agents.

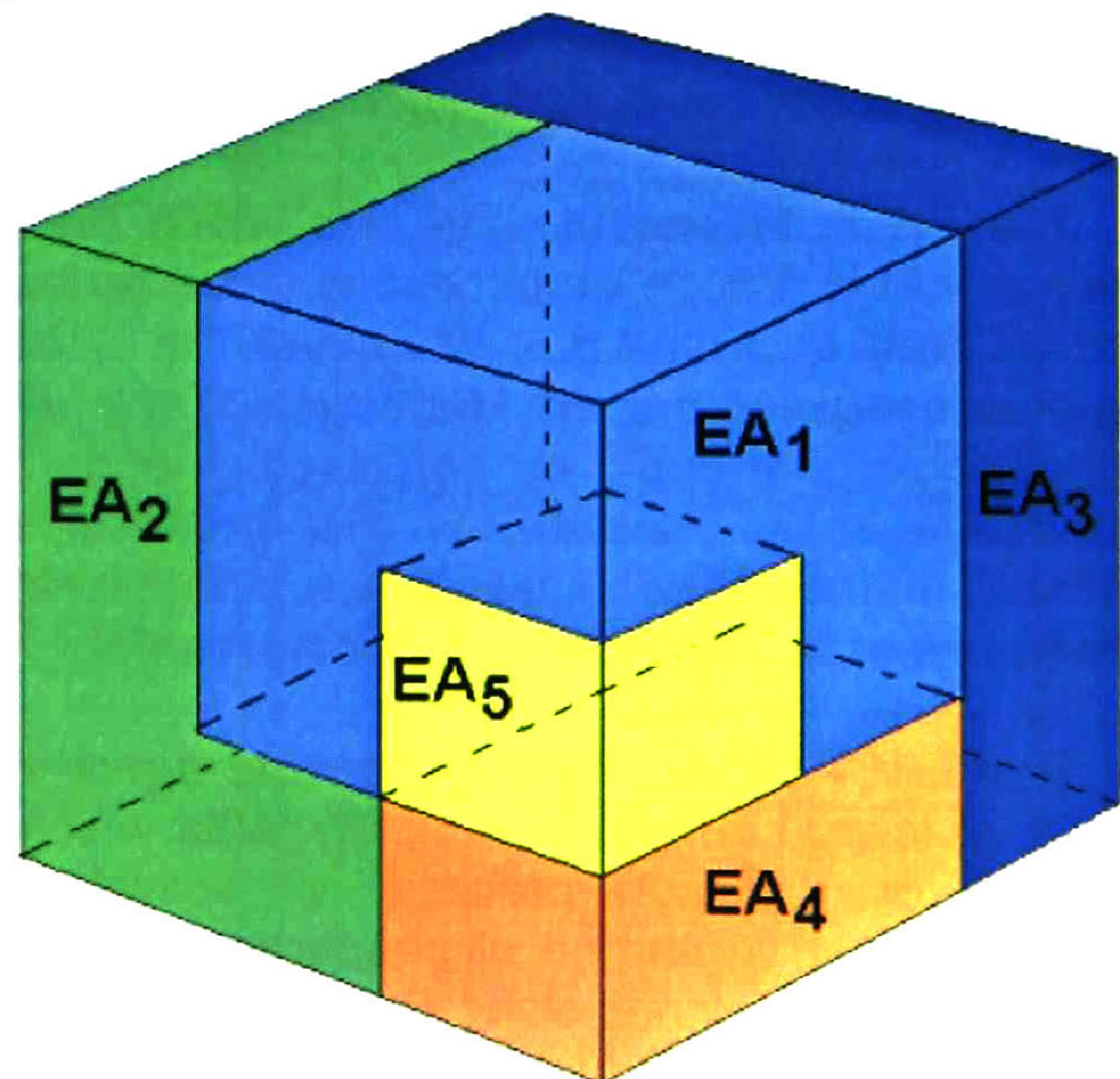


Figure 5-9. The Environment divided into regions of volume *x*

The distribution is adaptive, because the number of agents that represent the environment is calculated in function of the complexity of the environment for each time *t*. If the environment is complex, then we will have that the number of Environment Agents is big, otherwise is small. Our strategy offers the mechanism to increase or reduce the number of agents in function of the environment complexity.

We propose measuring the complexity of the environment through the class DistributionMap as illustrated in Figure 5-10. The DistributionMap (DM) specifies

the collection *cubes* that compose the entire environment as a 3D space. The DM has an administrator defined as the AUN (see section 4.6.2) of the corresponding EA. The DM establishes a default *cube count* for *x*, *y* and *z* coordinates in order to determine the corresponding bounds of every EnvironmentCube according to bounds specified by the target Environment.

Figure 5-10. The Distribution Map

DM represents a shared memory space to determine where all EnvironmentObjects are located. An EnvironmentCube determines which EnvironmentObjects are *members* or *neighbors* of the cubical space represented. An EnvironmentObject (EO) is located in one EnvironmentCube as a *member* and this is determined if the position of the EO is inside of the EnvironmentCube´s bounds, otherwise if the bounds of the EO overlap the EnvironmentCube it is considered a *neighbor*. Every agent intention takes some *time of response* to process it since such intention arrived to the EA´s MailBox (see section 4.5.4) until the EA sends updates to interested agents. In order to test distribution of the environment, an *increase of complexity* is declared simply if the *time of response*

for subsequent intentions increases three consecutive times. The Strategy is as follows:

At first, the entire environment is represented and administered by just one Environment Agent as illustrated in Figure 5-11. From now on, let the 3D environment be composed by a set of *x*=3 * *y*=3 * *z*=3 *cubes*. The *cube count* for each coordinate is determined by the end user.

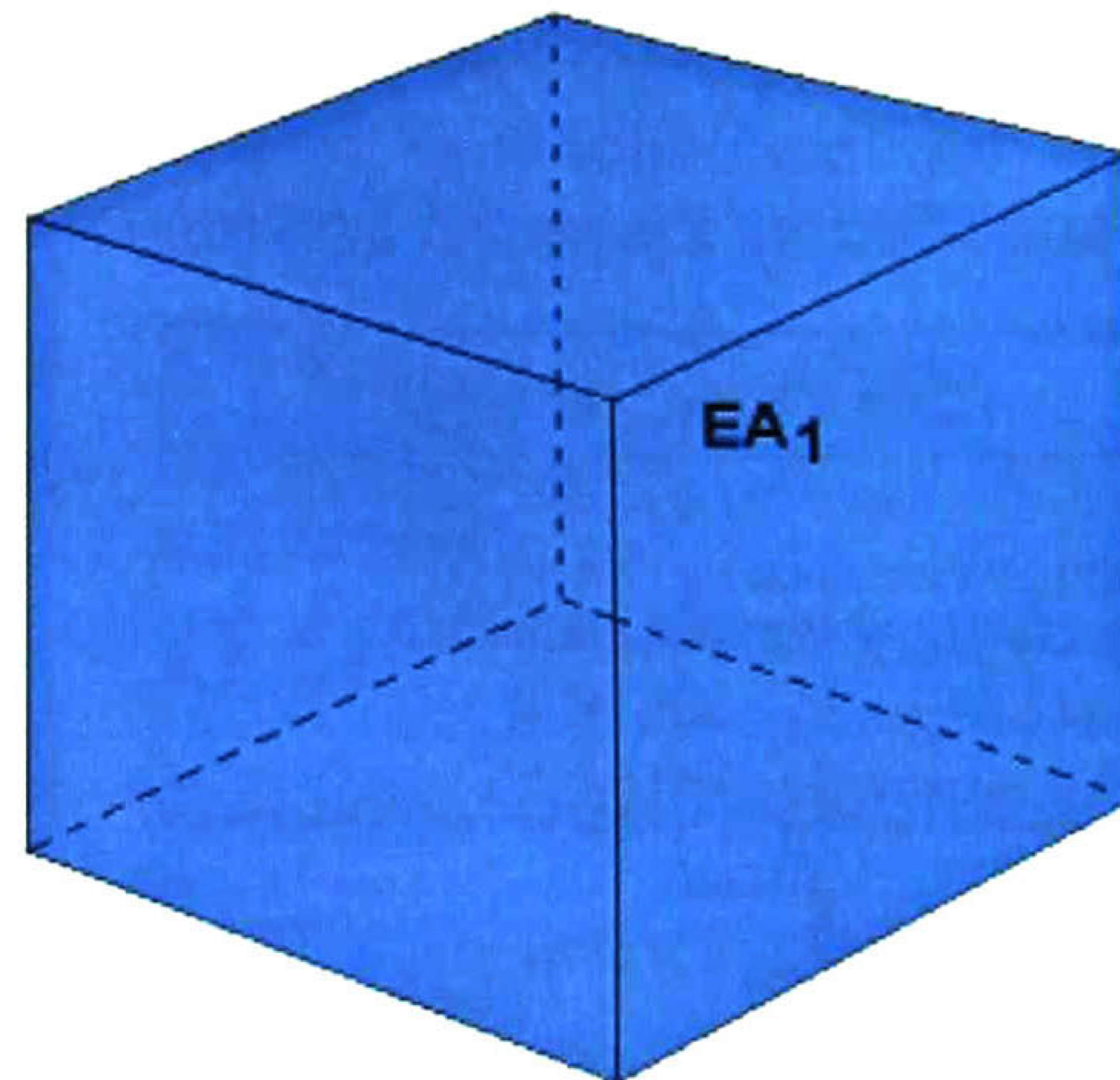


Figure 5-11. One EA administering the entire environment

When an Environment Agent decides to apply a dichotomy (cloning itself for another instance), this decision is based on an important increase of complexity it manages. Our solution resolves the problem for deciding where to generate a new Environment Agent and what information is provided to it. In Figure 5-12 EA_1 now administers just 8 *cubes* (see also Figure 5-3).

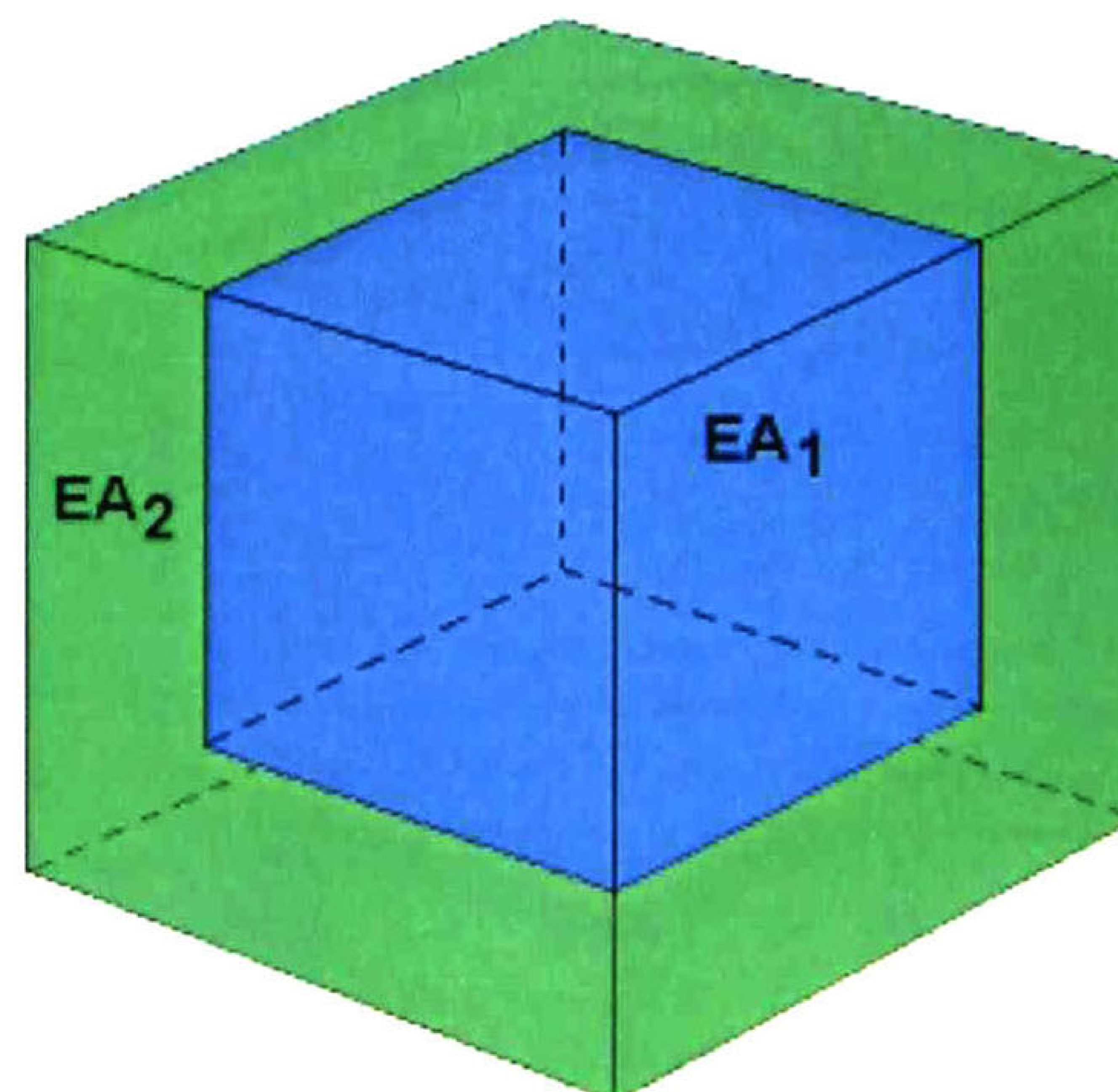


Figure 5-12. Dichotomy process on EA_1 creates EA_2

The Environment Agent will divide itself into sub-regions in function of the complexity of the environment for each time *t*. The complexity is measured according to the quantity of interactions that are made among entities with each other and between them and the environment. In Figure 5-13 EA_3 administers 9 *cubes* previously administered by EA_2.

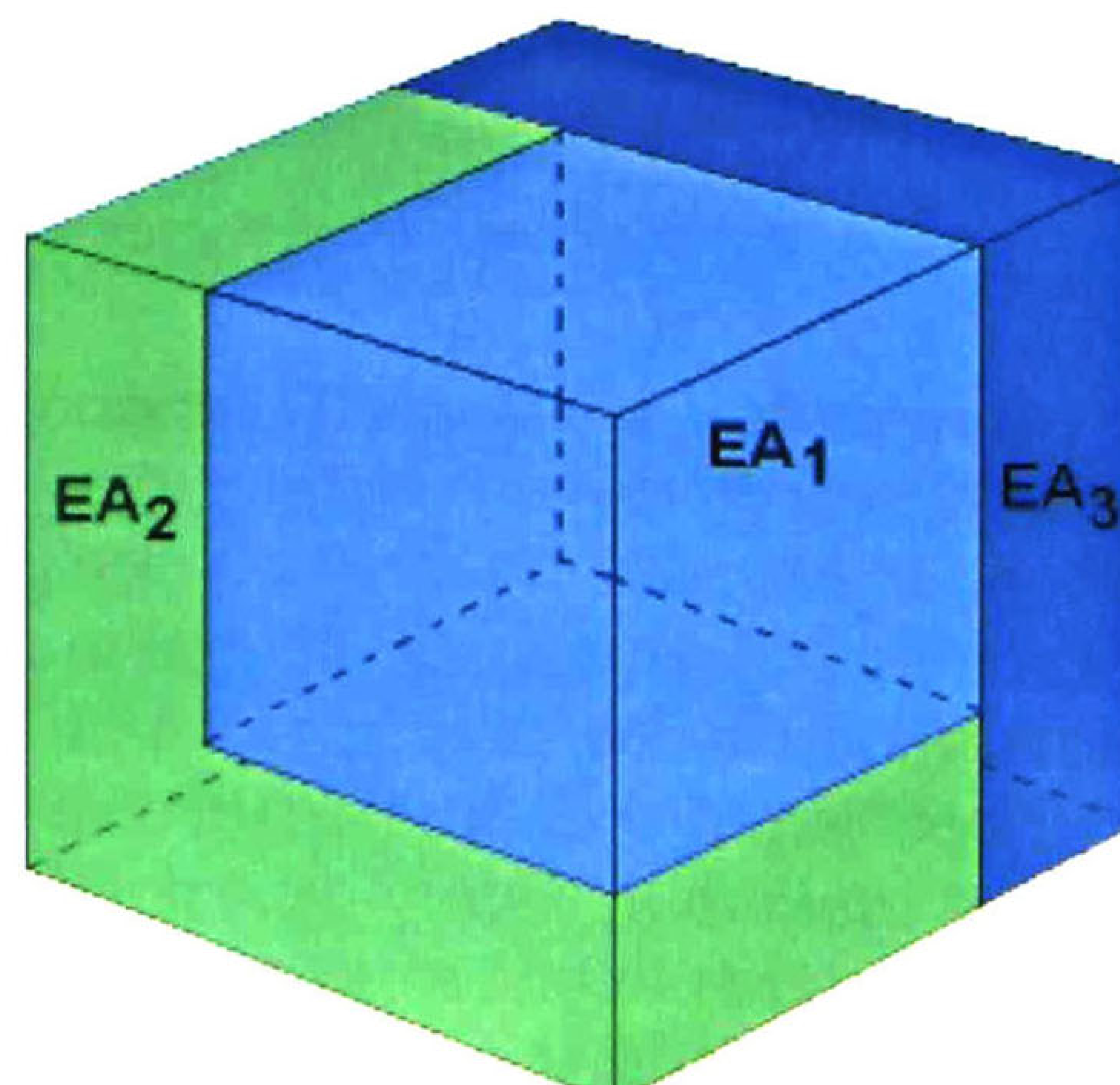


Figure 5-13. Dichotomy process on EA_2 creates EA_3

The quantity of agents will vary on time according with environment evolution. All agents will be able to communicate with each others to share border information. In Figure 5-14 EA_4 administers 2 *cubes* previously administered by EA_2.

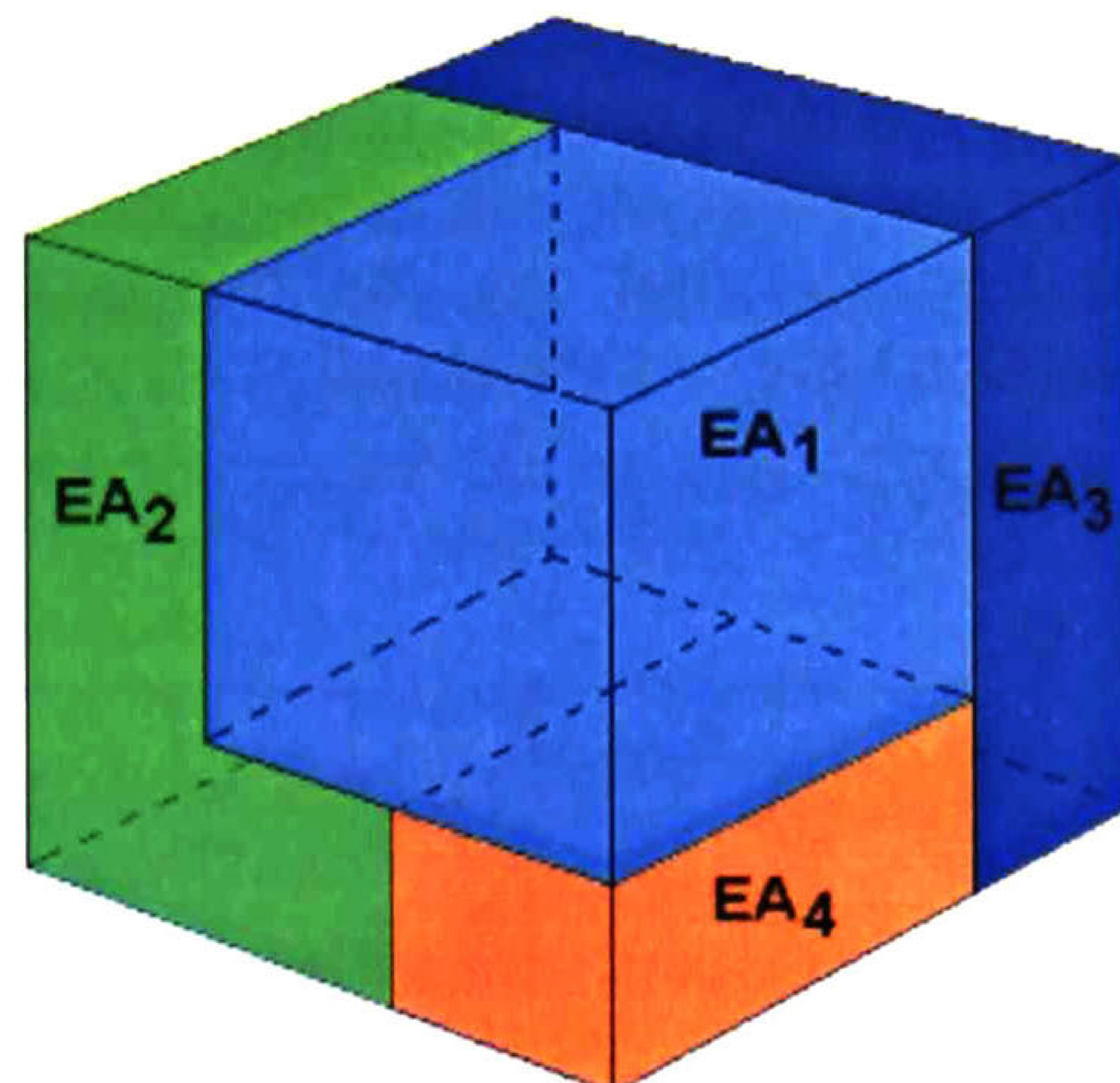


Figure 5-14. Dichotomy process on EA_2 creates EA_4

Distribution is done in function of the complexity of each part (a region *r*) of the environment. We will have more Environment Agents (resolving the bottleneck) in sections of the environment that contain more complexity and, a smaller number in sections with low complexity. In Figure 5-15 EA_5 administers one *cube* previously own by EA_1.

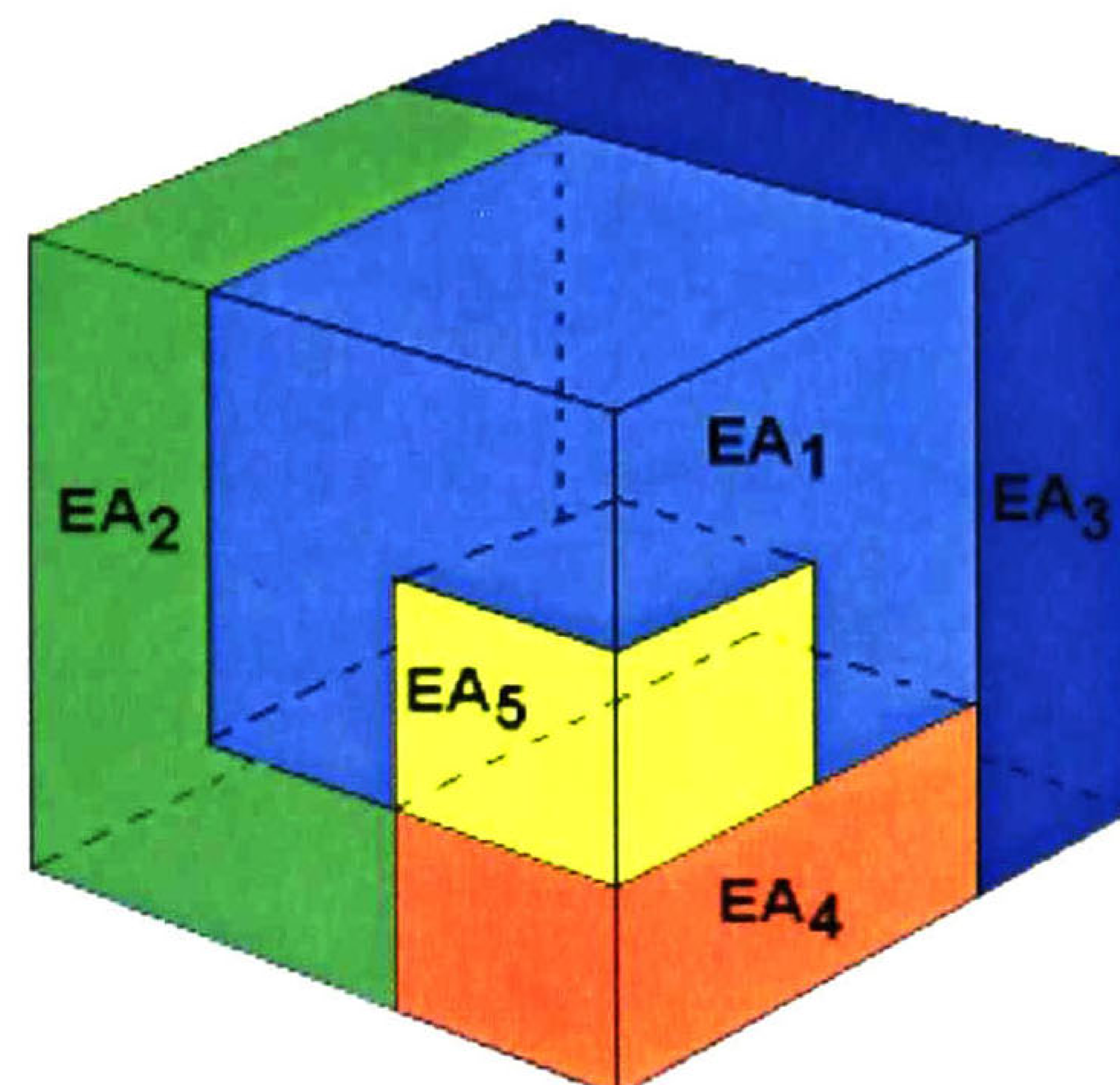


Figure 5-15. Dichotomy process on EA_1 creates EA_5

If complexity of some regions diminishes, two or more Environment Agents will agree to fuse their information (see section 5.4). Thus, one agent will represent two regions entirely. For example in Figure 5-16, EA_1 now represents previous EA_1 and EA_4 regions and, EA_2 represents previous regions EA_2 and EA_5. Regions do not have to stand next to each other.

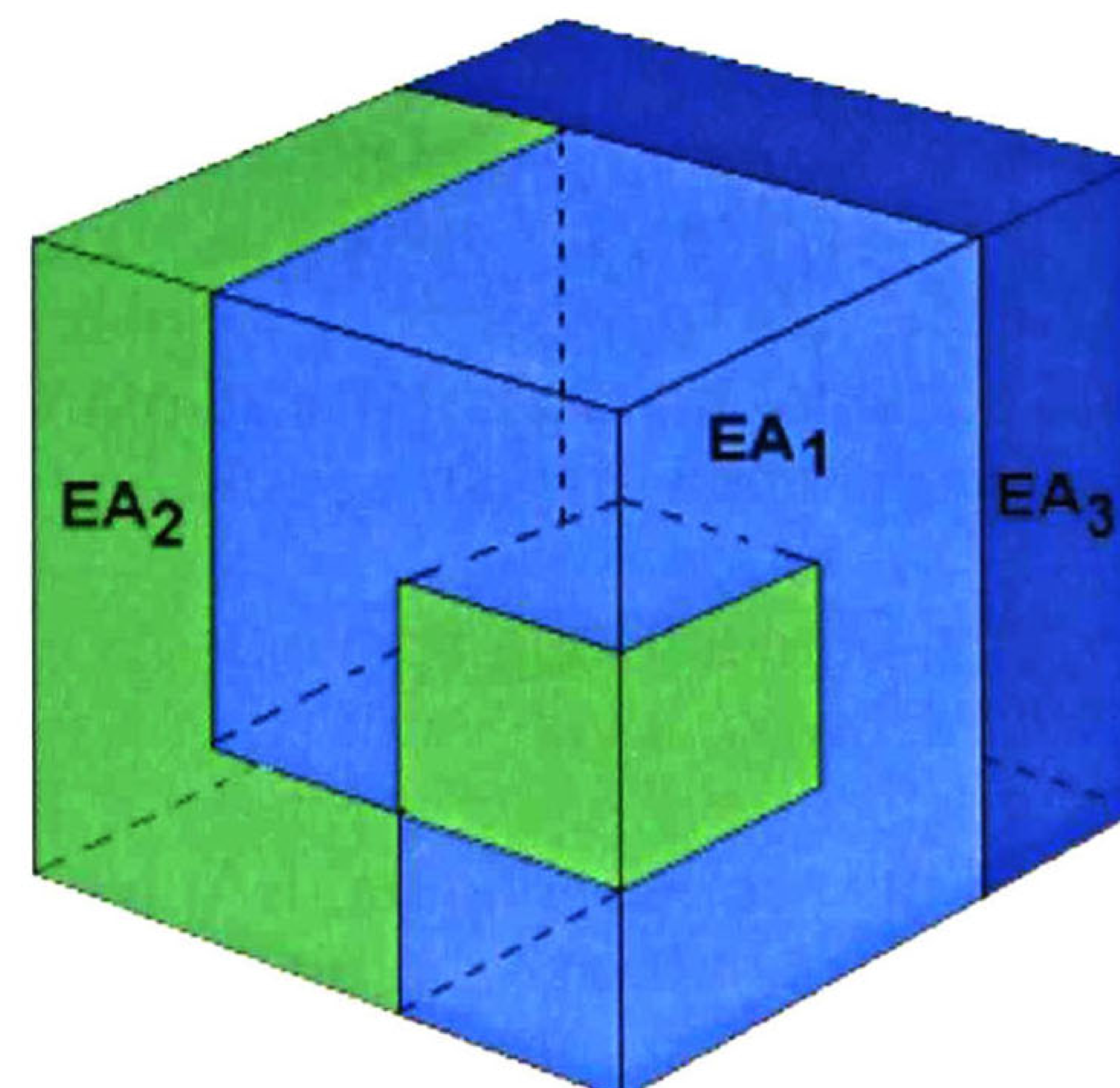


Figure 5-16. Fusion process of EA_1 with EA_4 and fusion of EA_2 with EA_5

As previously established, an increase in the complexity of the environment is declared, if the *time of response* of three consecutive intentions increases. Such increase of complexity leads to carry out a dichotomy process due to the following situation. Suppose that any intention takes 5 units of time to be processed and, three intentions arrive almost at the same time as illustrated in Figure 5-17

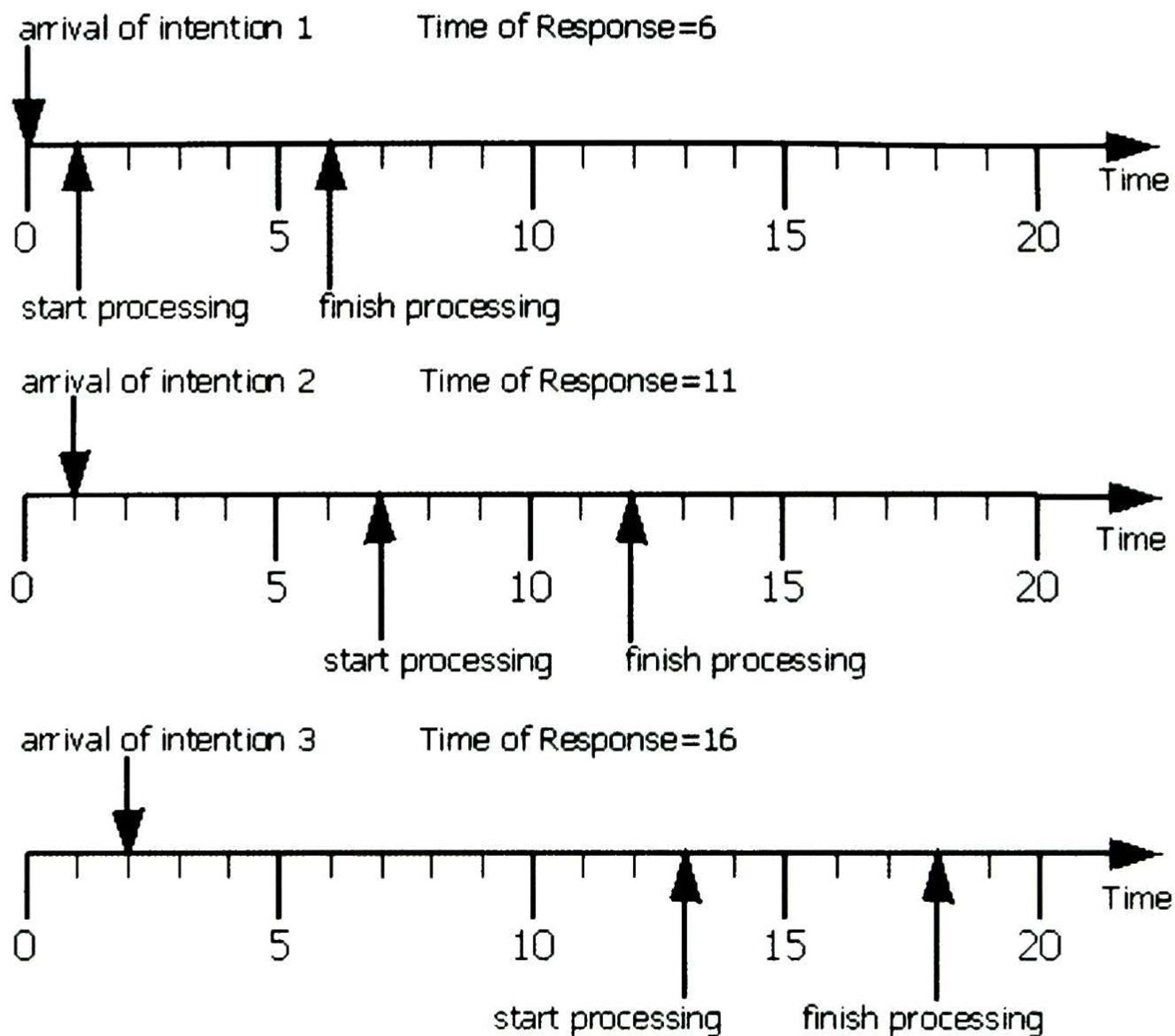


Figure 5-17. Three intentions with an increase of time of response.

In Figure 5-17 the time of response for these intentions is greater from the target agent perspective. Thus if the latter situation continues, some agents will not receive the environment update fast enough. Due to the cases of study (see chapter 6) available for tests, the increase of complexity is declared only considering the increase of *time of response*. In a more realistic simulation, we could also consider the average load of the EA, the amount of administered agents and the relationships between agents and objects.

Every EnvironmentCube specifies its administrator, so for every agent intention that the EA receives, it verifies if such agent is a *member* of any EnvironmentCube administered by the EA, otherwise the EA requests the platform to forward the intention to the corresponding EA in charge of such agent. This activity is analogous to distributed shared memory issues; the case when a page is requested in the local machine and such information is not located locally (see section 2.2.5), but instead of requesting the information, the current EA knows which EA owns the information and, the current EA proceeds to forward the intention.

An EO may be contained in one or more EnvironmentCubes, in one as a *member* and in the others as a *neighbor*. If an EA processes an intention that updates one of its member EnvironmentObjects, and such EO is contained as a neighbor of one or more EnvironmentCubes that are not administered by this EA, then it sends an update of that EO to other Environment Agents to cause that they also validate the intention and determine the corresponding changes in their corresponding environment partitions. The latter is analogous with the sharing of border information between cells (see section 2.3.1).

The *time of service* is the amount of time that an EA is giving service since launched or has recently participated in a dichotomy process of a fusion process (see section 5.4). A *decrease of complexity* is declared by an EA when it detects that only the 10% of the *time of service* or less has been dedicated to agent intentions. A decrease of complexity is also declared, if all of the EA's members Environment Objects become neighbors, that is, they moved (or were moved) to EnvironmentCubes administered by other Environment Agents long time ago.

5.4 Dynamic Adaptation Policies

Initialization process of an EA:

1. EA is instantiated.
2. EA sets its flag *adaptationInProgress* as true.
3. EA requests platform to registerAgentUniqueNameForLookup() to be localizable as a unique entity (see section 4.6.2)
4. EA joins the Environment Agent group through invoking the operation joinGroup (see section 4.6.3).
5. EA receives environment description and optionally the shared memory state. The first EA created for an environment will not receive a shared memory state, because in the beginning this EA will represent the entire environment
6. EA requests to the Context Module the necessary data related to Environment Objects according to the received environment description, and stores such data in the ContextCache in order to calculate agent intentions consequences.
7. EA starts measuring the *time of service*.
8. EA sets its flag *adaptationInProgress* as false.

DistributionMap is a class accessible by the threads: EA when executes behavior(), AgentListener when executes messageReceived() and Fuser when carries out the fusion process. Such threads right before trying to access to DistributionMap test the flag *adaptationInProgress* and suspend their execution while that flag is true in order to guarantee mutual exclusion in the DistributionMap for readings and writings. After the thread realizes that such flag is false it sets the

adaptationInProgress flag as true. After such threads finish accessing to the DistributionMap, they set the *adaptationInProgress* flag as false.

The EA processes agent intentions sequentially as they arrive into MailBox. The EA is processing an agent intention if and only if it could set the flag *adaptationInProgress* as true.

When an EA is launched, its AgentListener (see section 4.6.3) may invoke its messageReceived() method, as long as the EA is registered at platform; this method is used to receive ACLMessages from any module or agent and, to store agent intentions into a queue in the ContextCache.

After ContextCache queues an intention, it notifies the EA. After the EA is notified, it processes agent intentions according to rules read from the ContextCache. The EA is initialized with the flag *adaptationInProgress* set as true (see Figure 5-8) in order to its behavior waits until the environment description is received by the AgentListener.

For every intention the EA processes, after calculating the corresponding change in the environment representation, it calculates the *time of response* of that intention measured since such intention arrived to MailBox (see sections 4.5.4 and 4.6.3) until update messages were sent to all interested agents.

Besides, for every intention processed, the EA stores on the DistributionMap´s LoadTable for every updated object the *intention time of service*, measured since that intention was obtained from the queue until update messages were sent. Every updated object might be either a member or a neighbor of the EnvironmentCube. The LoadTable accumulates such *intention time of service* to the one previously stored if that already existed. The LoadTable´s data for every object will be considered as the object´s load. The EA also stores the *intention time of service* in the Fuser, in order to measure the *percentage of time of service* used.

After processing every intention, the EA compares the *time of response* with the previous one, and when it detects that such readings increased three consecutive times and, if this EA administers two or more EnvironmentCubes which members have load, then it concludes that the *upper bound limit* of *complexity* has been reached and starts a dichotomy process.

The load of all objects is used to calculate complexity of different combinations of sets of EnvironmentCubes when it is required to apply a dichotomy in order to find a combination of two exclusive sets with the closest to equalize workload, forming a Partition (see Figure 5-10) of two regions, which are subsets of exclusive EnvironmentCubes (see Figure 5-3). The load of an EnvironmentCube includes both member and neighbor objects. The load of all objects in this EA is also used to notify other Environment Agents the average region load of this EA accumulating the load of all objects and dividing them by the *time of service*.

After processing every intention and if a dichotomy process was not started, then the EA verifies if the *percentage of time of service* is less than 10% and, if there are more Environment Agents running then the EA concludes that it has reached the *lower bound limit* and starts a fusion process. The EA also reaches the *lower bound limit* when it realizes that it administers EnvironmentCubes with no members in them for a determined period of time (which for tests such period is 60 seconds of *time of service*) and, that there are more Environment Agents running.

Dichotomy process (*upper bound limit*):

Let x and y be regions. Let EA_x be the EA in charge of region x. Let a partition be a set of Environment Cubes. At first x will exist and y will exist later.

1. EA_x´s behavior realizes that has reached the *upper bound limit*, measuring *time of responses* since its instantiation or its last dichotomy process. EA_x´s had previously set the flag *adaptationInProgress* as true (as stated before).
2. AgentListener of EA_x continues queueing intentions into ContextCache.
3. EA_x's behavior launches a thread Splitter.
4. Splitter requests platform to create another EA. This new EA will be in charge of region y so it will be called EA_y.
5. EA_y will be loaded on the machine with the lowest workload (see section 4.5.1).
6. Splitter waits to be notified of EA_y´s AUN (see section 4.6.2).
7. EA_x's behavior determines the best partition according to the load for every object that EA_x administers.
8. EA_x's behavior prepares part of an ACLMessage to be sent to EA_y, establishing the Environment Partition to be sent.
9. EA_x's behavior deletes all information about the Environment Objects that are not members neither neighbors of any of the EnvironmentCubes that EA_x's administers.
10. EA_x's behavior waits until Splitter can provide EA_y´s AUN.
11. EA_y carries out the initialization process described previously.
12. Splitter receives back the AUN of EA_y.
13. EA_x's behavior establishes in the DistributionMap that the administrator of the second partition is the AUN received.
14. EA_x serializes the shared memory state of the DistributionMap and stores it as binary content in the ACLMessage to be sent.
15. EA_x's requests platform to send the ACLMessage to EA_y.
16. EA_x sends to the Environment Agent group the update for the DistributionMap.
17. The other Environment Agents update their corresponding DistributionMap.
18. EA_x sets the flag *adaptationInProgress* as false.

Any EA created after the first one will receive an environment partition, that is, a subset of the Environment Objects that its creator (another EA) owned.

In Figures 5-18 and 5-19 we depict the activities to determine the best partition.

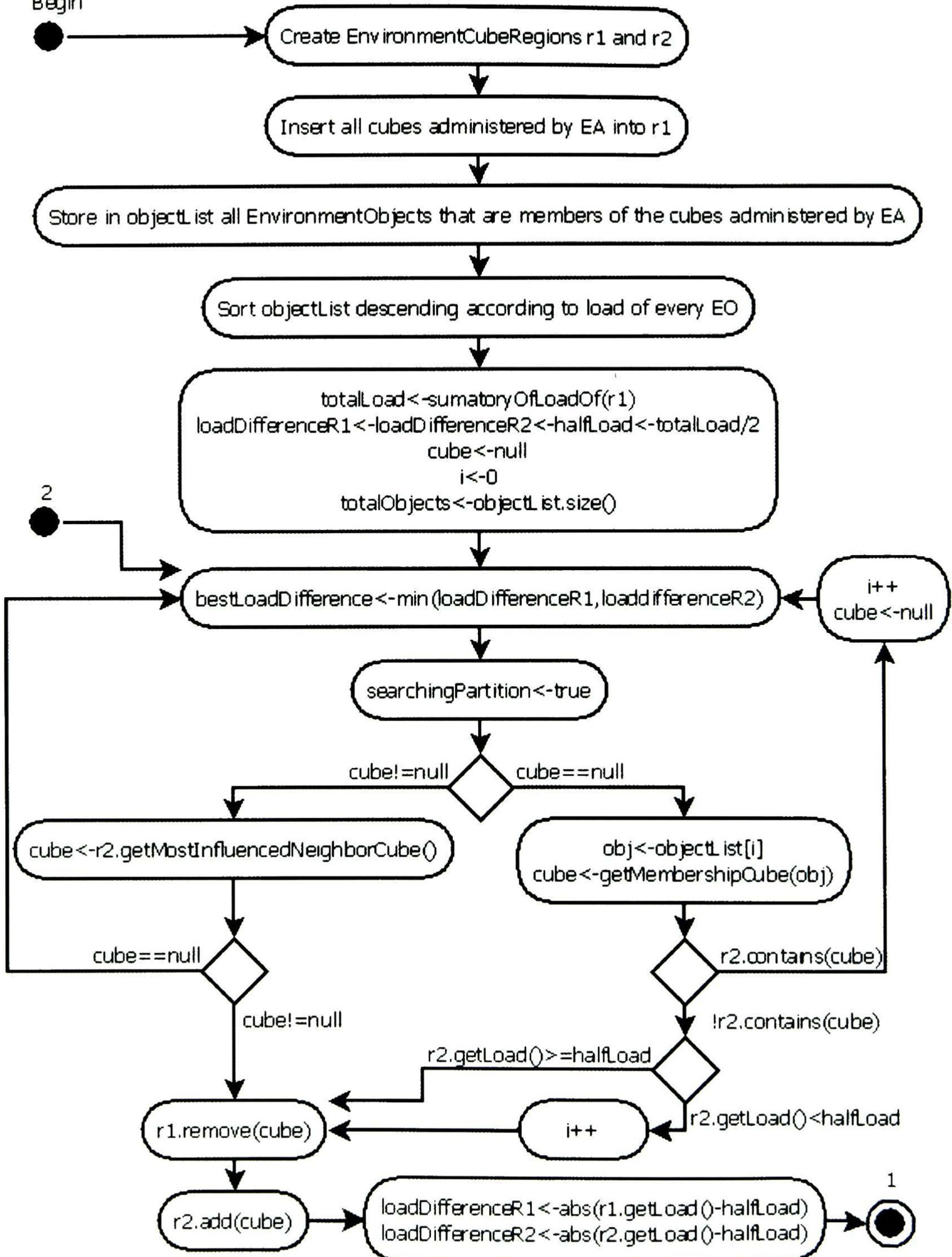


Figure 5-18. Activity diagram to determine the best partition during dichotomy process (part 1).

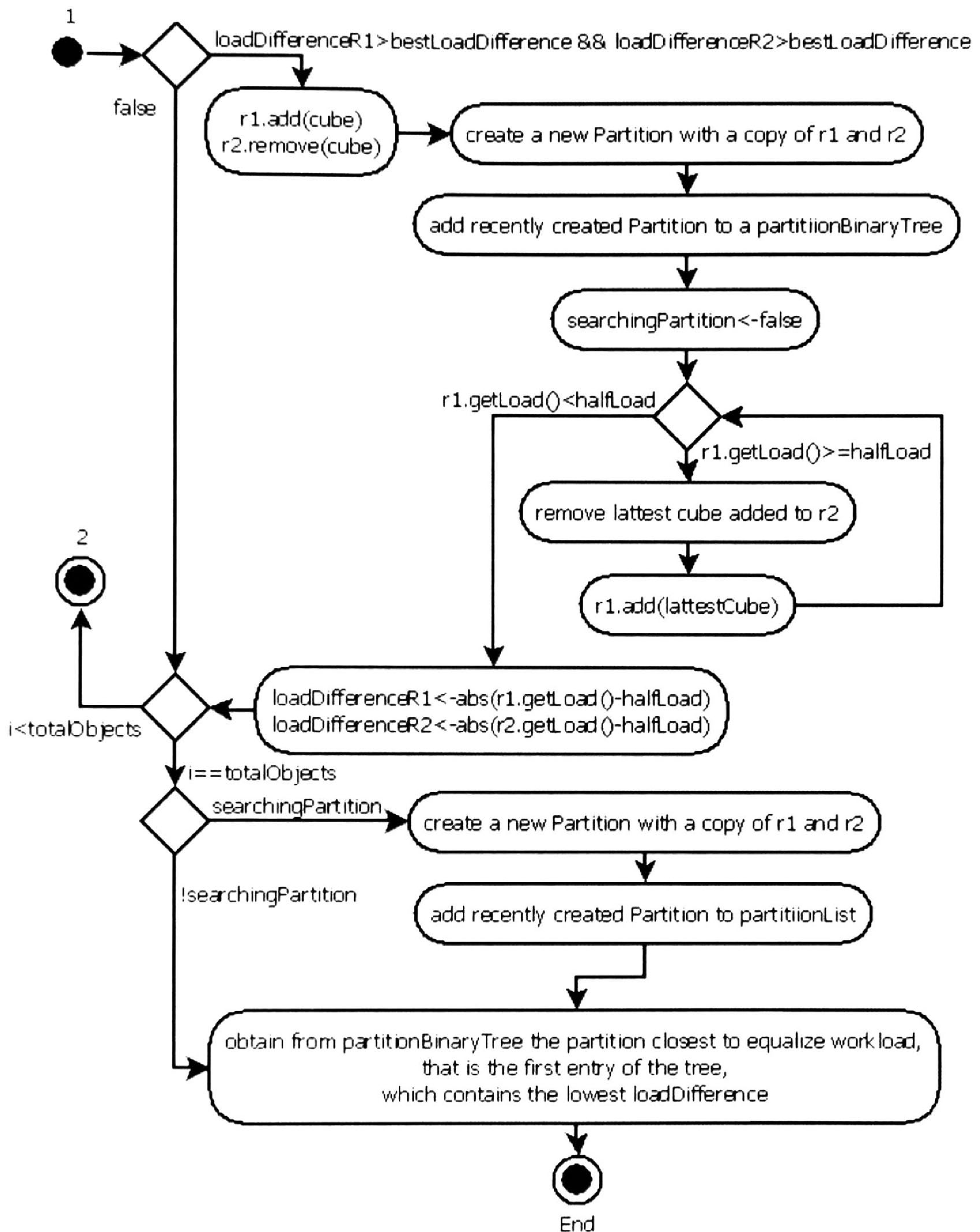


Figure 5-19. Activity diagram to determine the best partition during dichotomy process (part 2).

Let *w* and *z* be regions. For every intention that EA_z receives, it verifies whether the agent that provided the intention is one of the Environment Objects that this EA_z administers, otherwise EA_z queries its DistributionMap to find the EA_w in charge of such agent and then requests platform to forward such intention to EA_w.

Fusion process (*lower bound limit*):

Let *p* and *q* be regions.

1. EA_p realizes that has reached the *lower bound limit*. EA_p´s had previously set the flag *adaptationInProgress* as true (as stated before).
2. EA_p´s behavior launches a thread Fuser that will communicate with another EA on behalf of EA_p.
3. EA_p sets the flag *adaptationInProgress* as false to let Fuser access the DistributionMap.
4. AgentListener of EA_p continues delivering agent intentions to EA_p.
5. EA_p continues processing agent intentions.
6. Fuser announces (sending a group message) to other EAs its intention to fusion with another EA, managing a Contract Net Protocol (see section 2.3.4).
7 Every one of the other Environment Agents may or may not submit a bid indicating its region load according to the following criteria, that is analogous to the distributed algorithm of mutual exclusion [TANENBAUM]. This mechanism allows the fusion process be carried out by only one EA at a time, in order to avoid two Environment Agents choosing each other to fuse and, both disappear from de environment:
   a) An EA submits its bid if it does not require to fusion with others.
   b) If an EA is executing its Contract Net Protocol and receives an announce from another EA, then compares its AUN versus the other EA´s AUN and, if the local AUN is before the remote AUN then, then this EA sends its bid, otherwise does not respond.
   c) If the EA sends a bid, then proceeds to cancel its fusion process and, only after step 14 is executed, this EA could carry out a fusion process if determines it as necessary.
8. After receiving all bids, EA_p determines the EA_q with the lowest region load.
9. Fuser sets the flag *adaptationInProgress* as true in order to avoid that EA_p´s behavior modifies the DistributionMap.
10. Fuser sends a proposal to EA_q to carry out a fusion attaching its Environment Partition, working this as a sender initiated negotiation with work attached (see section 2.2.2).
11. When EA_q receives the proposal it updates its DistributionMap and its environment representation according to the received Environment Partition and the shared memory state. Such proposal is always accepted.
12. After updating its environment representation, EA_q sends back to EA_p a confirm message.
13. Fuser sends to the Environment Agent group the update of the DistributionMap.
14. The other Environment Agents update their corresponding DistributionMap.
15. Fuser updates its DistributionMap establishing EA_q as the new administrator of EA_p´s EnvironmentCubes
16. Fuser sets the flag *adaptationInProgress* as false to let EA_p´s behavior to continue.

17. EA_p requests platform to forward to EA_q every intention queued.
18. When EA_p receives the confirm message, it requests the platform to deactivate its MailBox in order to do not receive more intentions.
19. Subsequent intentions that arrive to the EA_p´s machine will be rejected using TA packets (see section 4.5.5).
20. When all intentions were forwarded, EA_p requests platform to deregister it.
21. After EA_p was deregistered, if any agent sends intentions to this EA, such messages will be rejected using AU packets causing that the client machine localizes another Environment Agent (see section 4.5.5).

5.5 Summary

We proposed an administration of the environment based on special autonomous agents we call Environment Agents. These agents represent only one part of the Environment. To alleviate the bottleneck problem, the proposed solution in this work distributes the environment (the scenario and the context) adaptively in a set of Environment Agents representing both modules Scenario (MS) and Context (MC).

Thus, an Environment Agent (EA) is in charge of administering a region r consisting of a MS partition and a MC cache.

The Scenario Module contains a collection of Environment Objects and a cubical space description.

The GenericEnvironment represents a 3D environment and it is composed by a collection of GenericEnvironmentObjects and, provides basic operations to manage any designable environment.

The GenericEnvironmentObject (GEO) specifies an object category as one of either an Avatar or an Object; where an Object would be a graphical virtual entity and an Avatar would be an Object with behavior administered by an agent.

A Cube determines the bounds of a component of the graphical environment. Each GEO belongs to one EnvironmentCube as a *member* and may be part of another EnvironmentCube as a *neighbor*.

The EnvironmentAgent is an Agent in charge of administering the Scenario of the virtual environment and also in charge of validating all permitted changes to the environment according to rules established in ContextCache that is a cached copy of a subset of the Context Module information.

Each Environment Agent represents a partial volume x of the environment. All Environment Agents administering the whole environment are perceived as a single entity by other agents

An Environment Agent is dedicated to processing agent intentions. At the same time it is aware of the upper and lower bound of complexity (computed with the *time of response* and the *time of service*) verifying the condition of fusion or its symmetric dichotomy.

We proposed measuring the complexity of the environment through the class DistributionMap, which specifies the collection EnvironmentCubes that compose the entire environment as a 3D space and stores information about the workload of each EnvironmentObject modified as a consequence of an agent intention.

An Environment Agent may decide to apply a dichotomy process, by means of cloning. It divides its information and workload with its clone agent. Vice versa, if it finds itself almost or already idle, it may decide to fuse with another Environment Agent. This behavior is achieved in function of the complexity of the 3D environment.

Chapter 6

Case Study

In this chapter we present graphical results and metrics of the behavior of the implemented environment, by means of the execution of three cases and, using all developed components for GeDA-3D integrated through the platform.

6.1 Introduction

Given that there is no context module as a service from which receiving information to validate intentions, we use ViSCA Render and AVE-3D Render in order to simulate execution time for context validation.

In the first case study we use the ViSCA Rendering. Every agent generates various intentions as soon as it can, according to its memory about the environment state. The Rendering is configured to cancel the executing update on an avatar, if another update arrives for the same avatar. The Rendering sends updates to the Environment Agent periodically, that is, it does not send a reply to every instruction it receives to modify the environment. This Rendering previously managed prototype versions of the classes Environment and the Environment Objects; such classes were reengineered according to the designed classes GenericEnvironment and GenericEnvironmentObject (see section 5.2).

In the second case study we use the AVE-3D Rendering. Every agent generates sequential intentions, once the agent sent its intention, it waits until the corresponding update arrives. We updated the AVE-3D Rendering in order to it uses Environment and EnvironmentObject implementations to represent the virtual environment.

Every simulation starts when launching the middleware for GeDA-3D in one or more machines; see Figure 6-1. The view for the platform is optional.

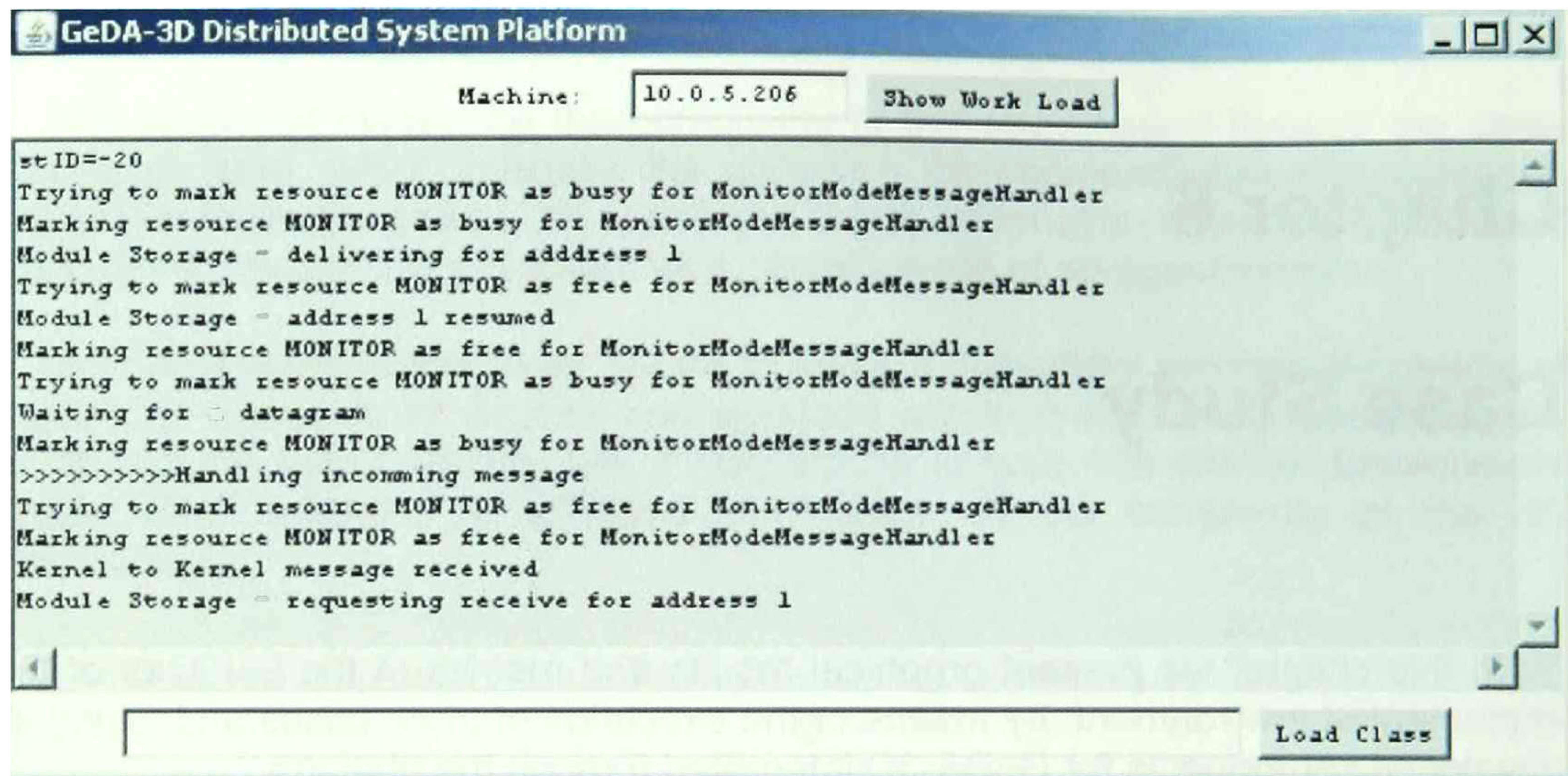


Figure 6-1. Start of the GeDA-3D Distributed System Platform

After that, the user can launch specific applications, either through the load class button or through the operating system console.

6.2 Prey Predator Ship Battle

We present here the case study on the work developed by [ZUÑIGA] [PIZA] and [AGUIRRE] for GeDA-3D

6.2.1 Illustrating the case without using dynamic adaptation policies

The simulation begins writing the script for the scene description and the scenario description using the Scene Descriptor as illustrated in Figure 6-2.

When the user presses the start button, the Environment Agent (EA) is started, it receives the scenario description in XML format, loads the environment classes designed to the current environment, and shows the amount of running "Environment Agents" the amount of "administered objects" each "objects load", the metrics for "time of service" "percentage of time of service used" "average region load" and the latest "time of response"; at the bottom it shows all "objects positions" in the virtual environment, showing the content of 9 of 27 Environment Cubes, using as rows the coordinate *y* and as columns the coordinate *x* because the EA determined that the environment was a two dimensional arrangement having all objects in the same coordinate *z*; all boxes show a label "second" which is the reference of time according to the local machine; see Figure 6-3.

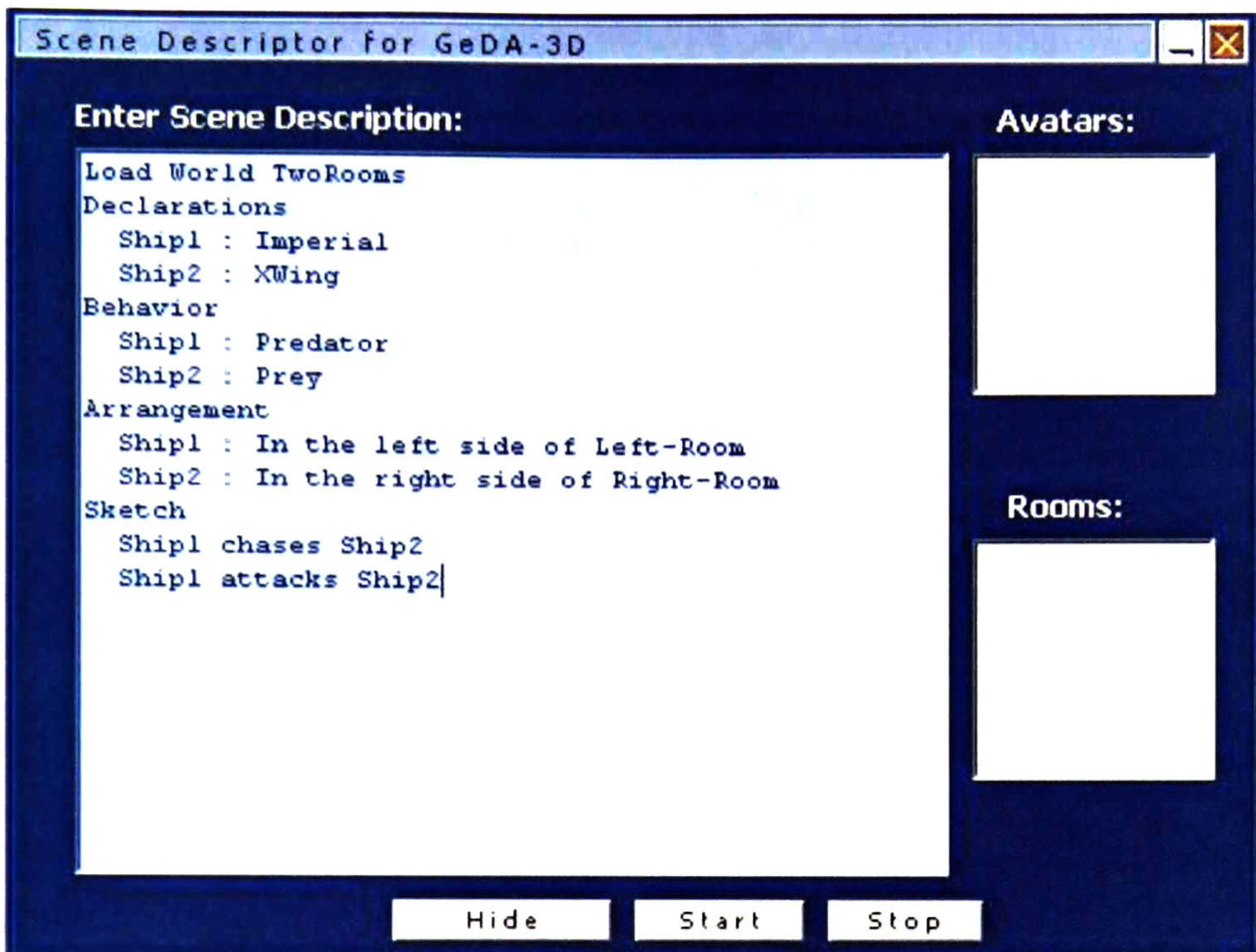


Figure 6-2. The scene and scenario descriptions for the first case study.

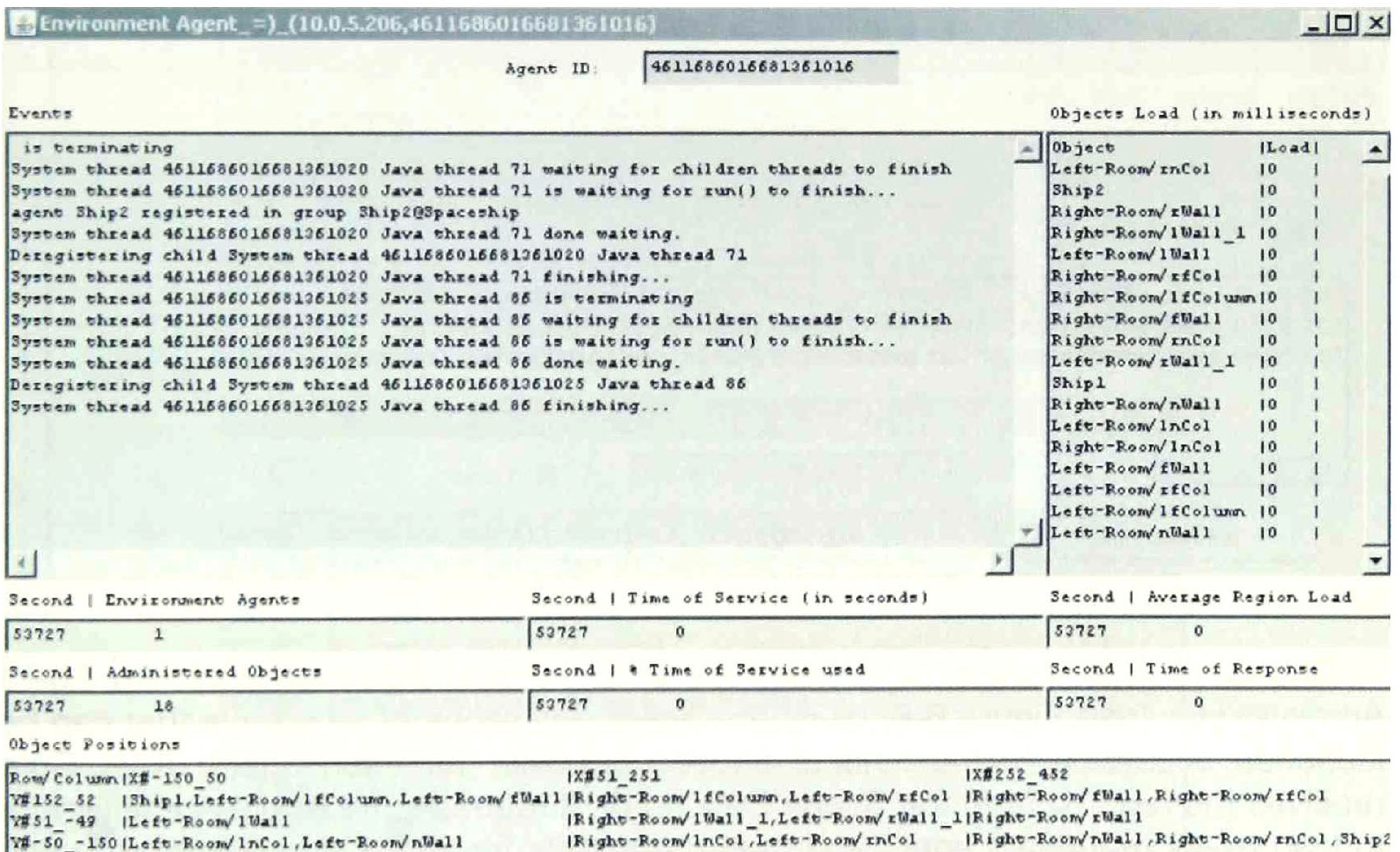


Figure 6-3. The Environment Agent at the beginning of the first case study.

For previous Figure 6-3 the center of the scenario is at (x=0, y=0, z=0). After the Scene Descriptor sent the scenario description to the Environment Agent, it shows

the scenario (see Figure 6-4) and, requests platform to communicate with the Core Module (see Figure 6-5). The Scene Descriptor requests Core to load the agents Ship1 (the blue ship) as a Predator and Ship2 (the yellow ship) as a Prey.

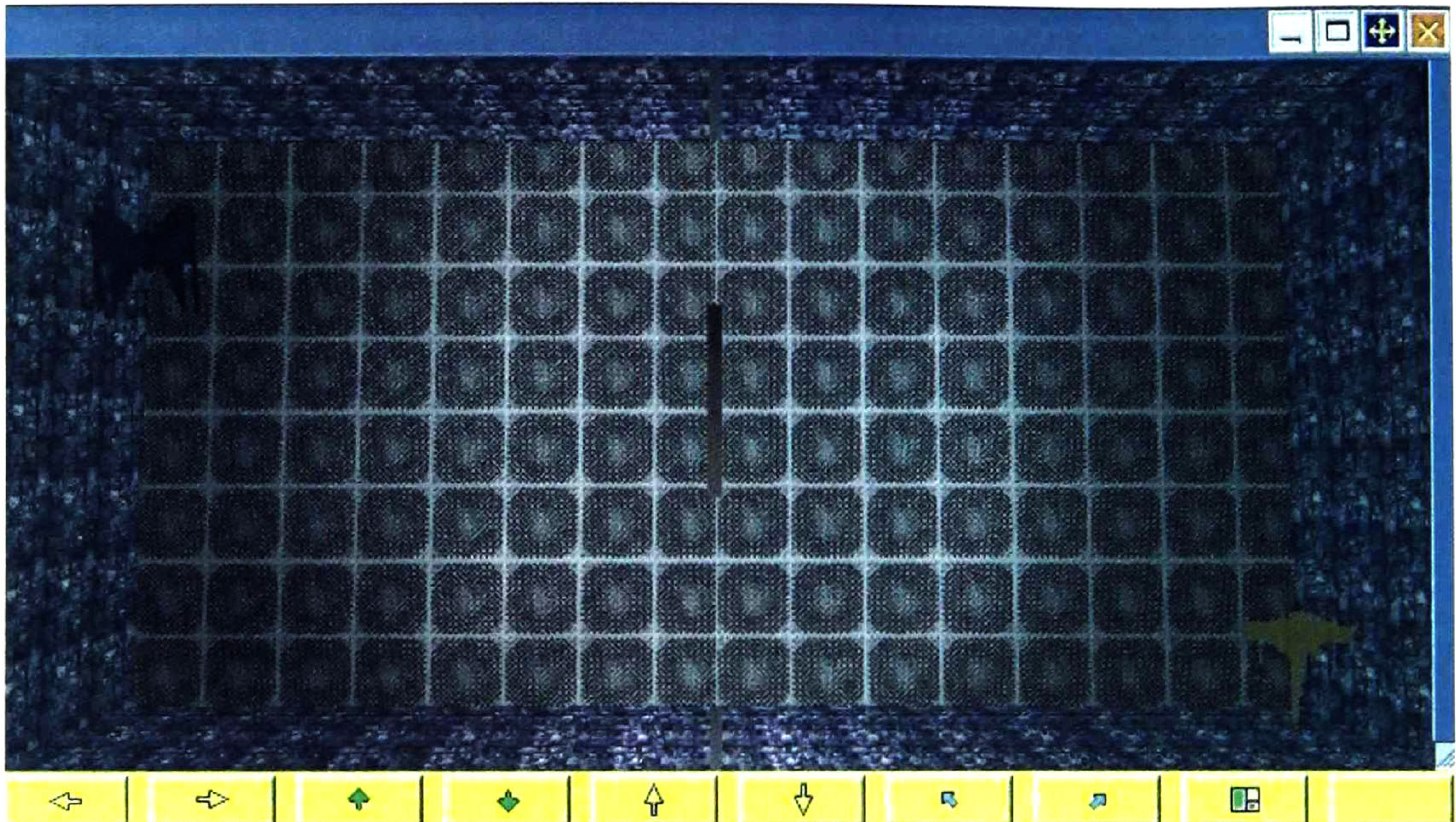


Figure 6-4. The Scenario´s state: at the beginning of the first case study.

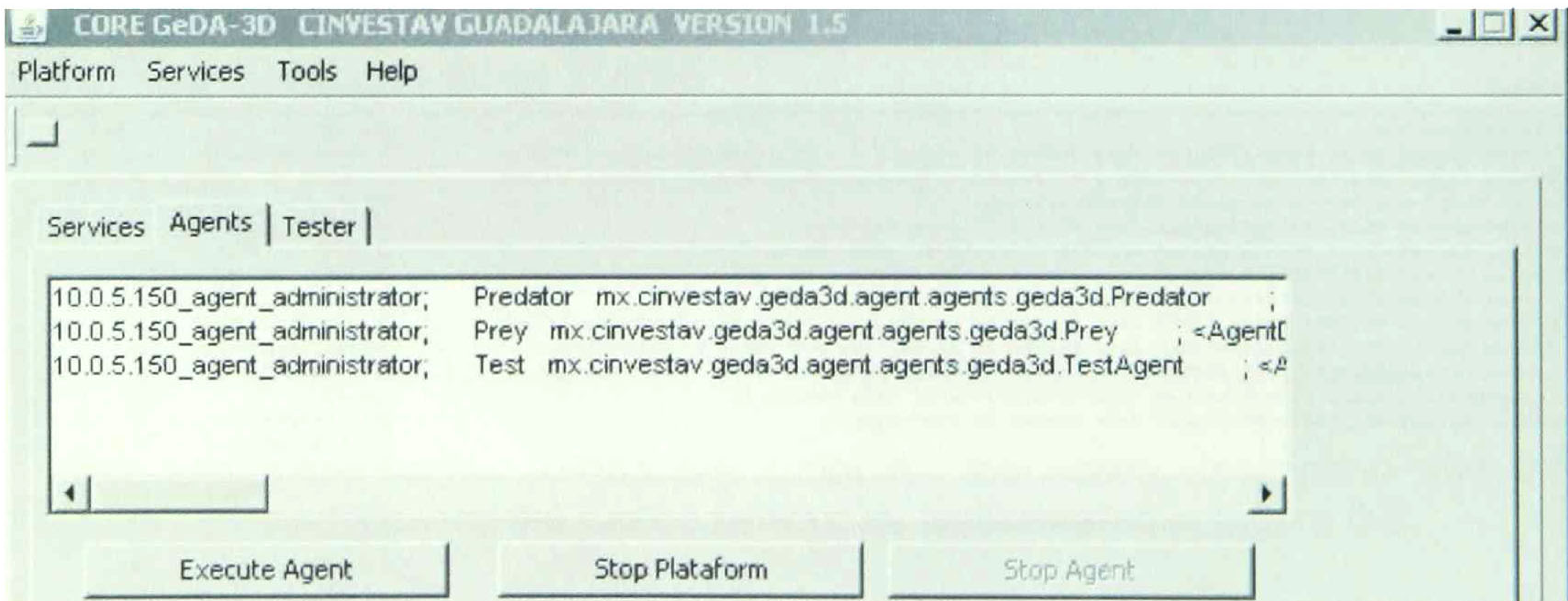


Figure 6-5. Core in the first case study.

The Core is in charge of locating all running Agent Administrators. An Agent Administrator (see Figure 6-6) owns the agent definitions of all agents that can be launched according to the kind of chosen behavior for each agent. After Core receives the request from the Scene Descriptor, it requests the Agent Administrator to load every requested agent and then the agents for Ship1 (see Figure 6-7) and Ship2 (see Figure 6-8) are loaded on the platform. After the EA loaded the environment classes according to the scenario description, it request platform to join agents Ship1 and Ship2 to a group created for every object near to their

avatars (including the group created for their own avatars). The EA sends to all agents only the objects that are around them and, for this case study, both ships know about the location of the other in order to Ship1 (the predator) chases Ship2 (the prey) and Ship2 moves from one place to another to make more difficult to be chased and attacked. By the way Ship1 uses a genetic algorithm to chase Ship2.

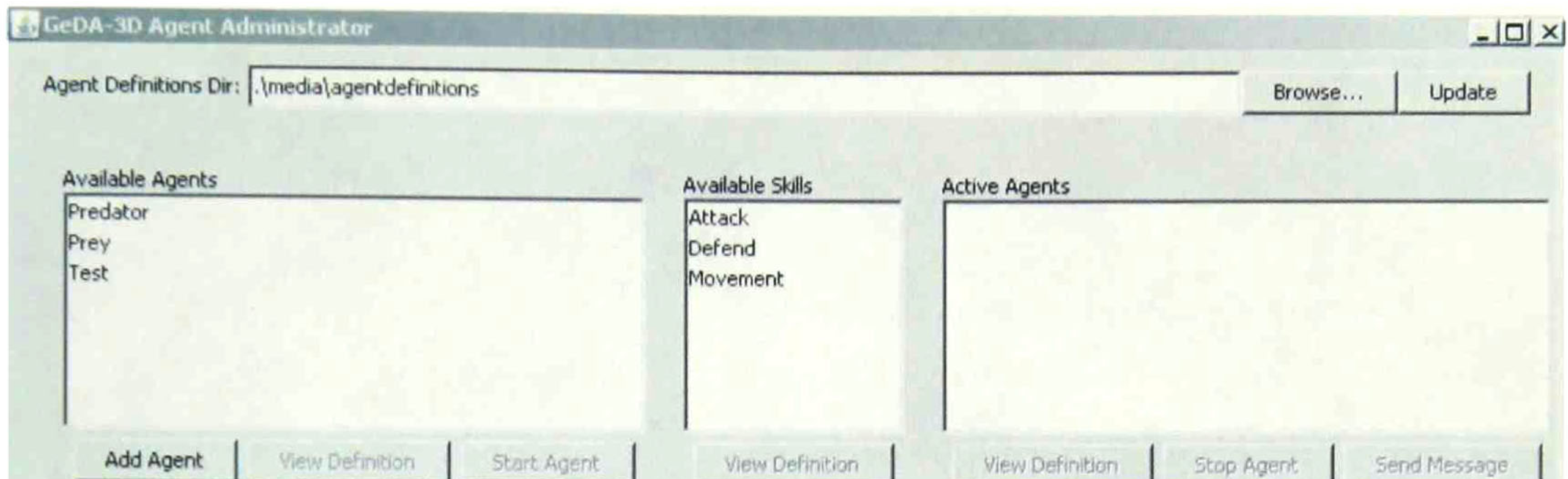


Figure 6-6. The Agent Administrator of the first case study.

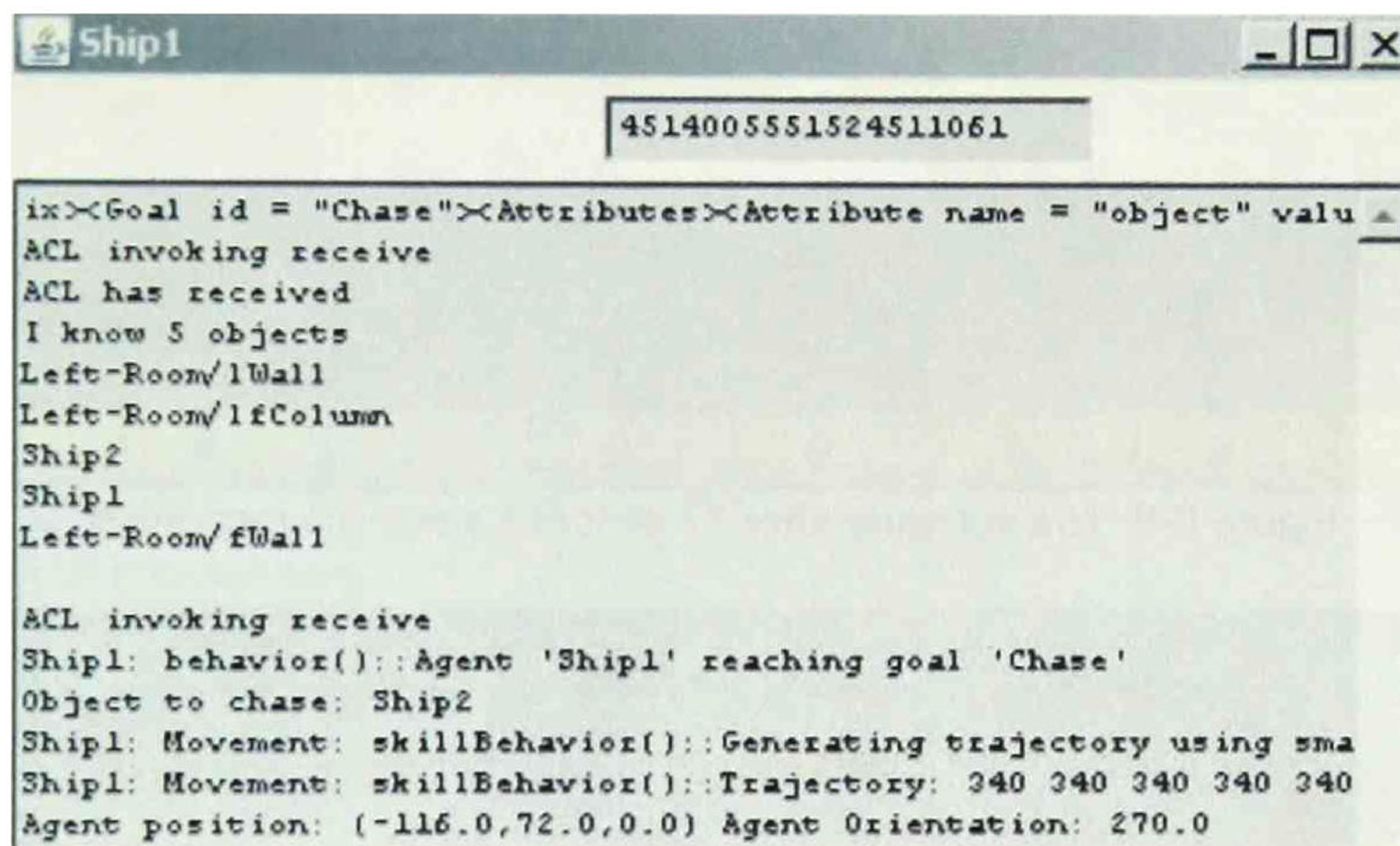


Figure 6-7. Ship1 at the beginning knows 5 objects of the environment.

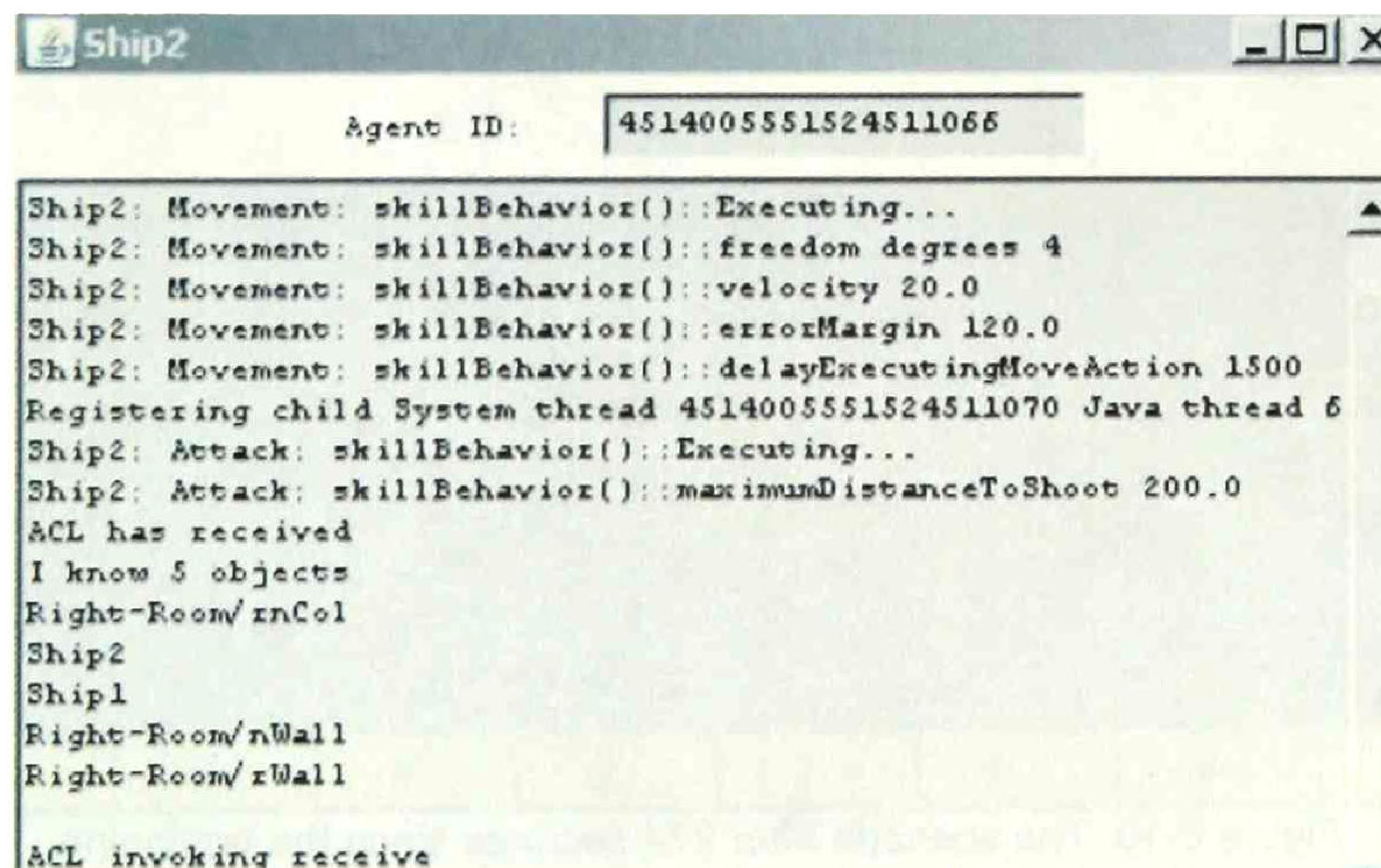


Figure 6-8. Ship2 at the beginning knows 5 objects of the environment.

All graphical interfaces but the Rendering and the Scene Descriptor are optional to be shown.

After both agents are ready, they send the Environment Agent their intentions to move their avatars; see Figures 6-9 and 6-10. The behavior of these agents was implemented by [ZUÑIGA].

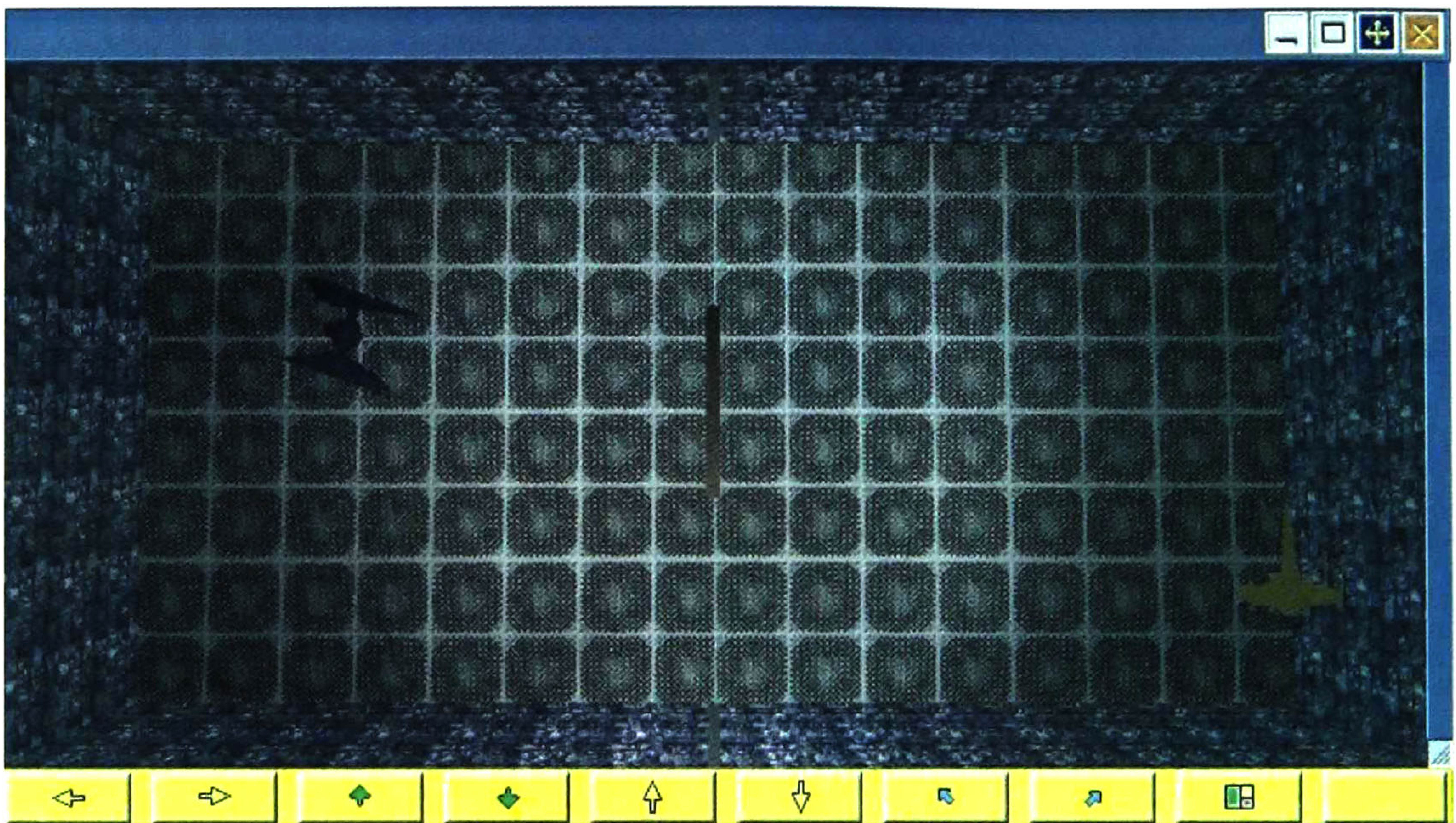


Figure 6-9. The scenario after 77 seconds since the beginning.

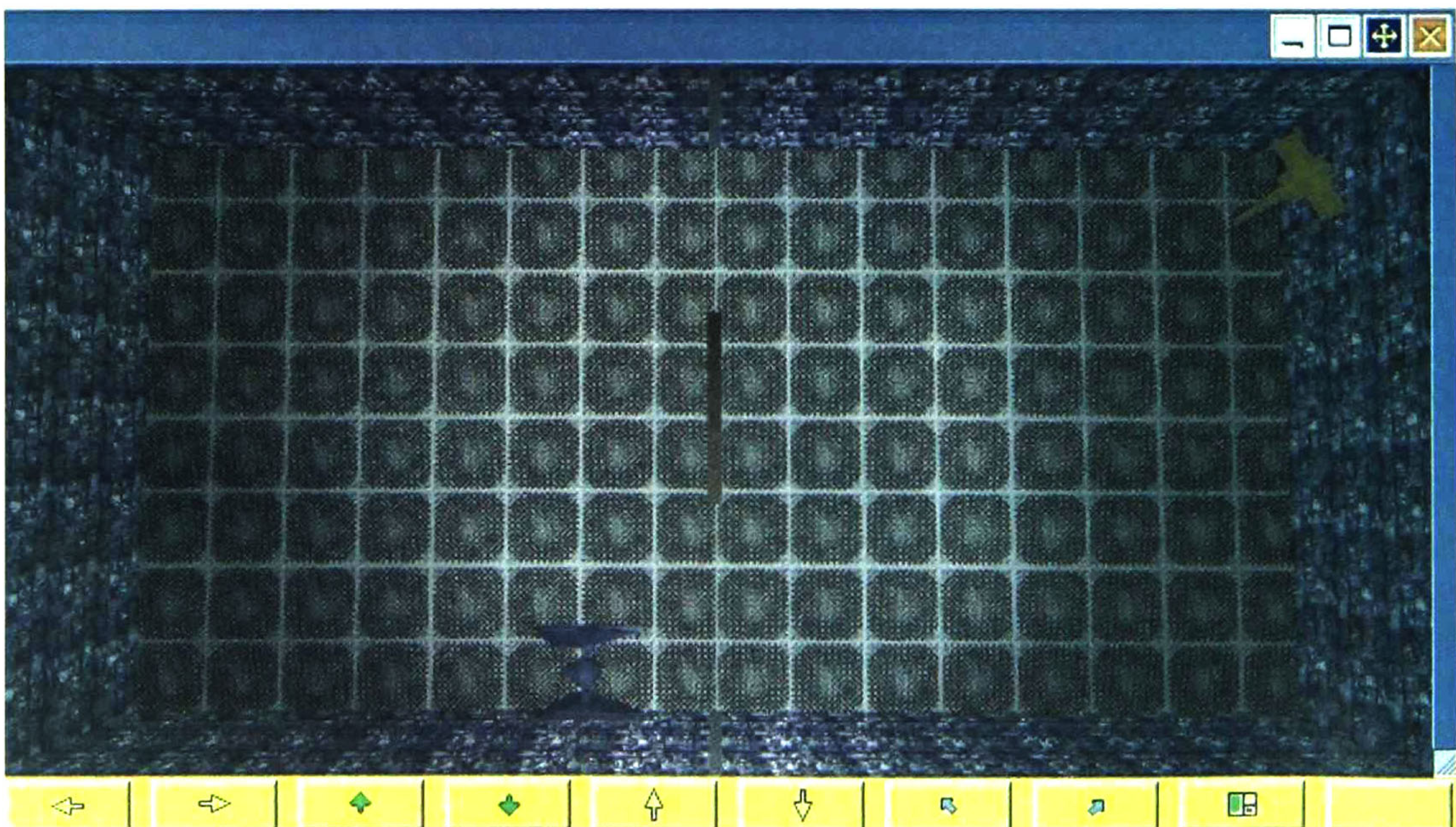


Figure 6-10. The scenario after 274 seconds since the beginning.

As long as the avatars move through the environment, the Environment Agent notifies to their corresponding agents, their current state and the state of objects around each one as illustrated in Figures 6-11 and 6-12.

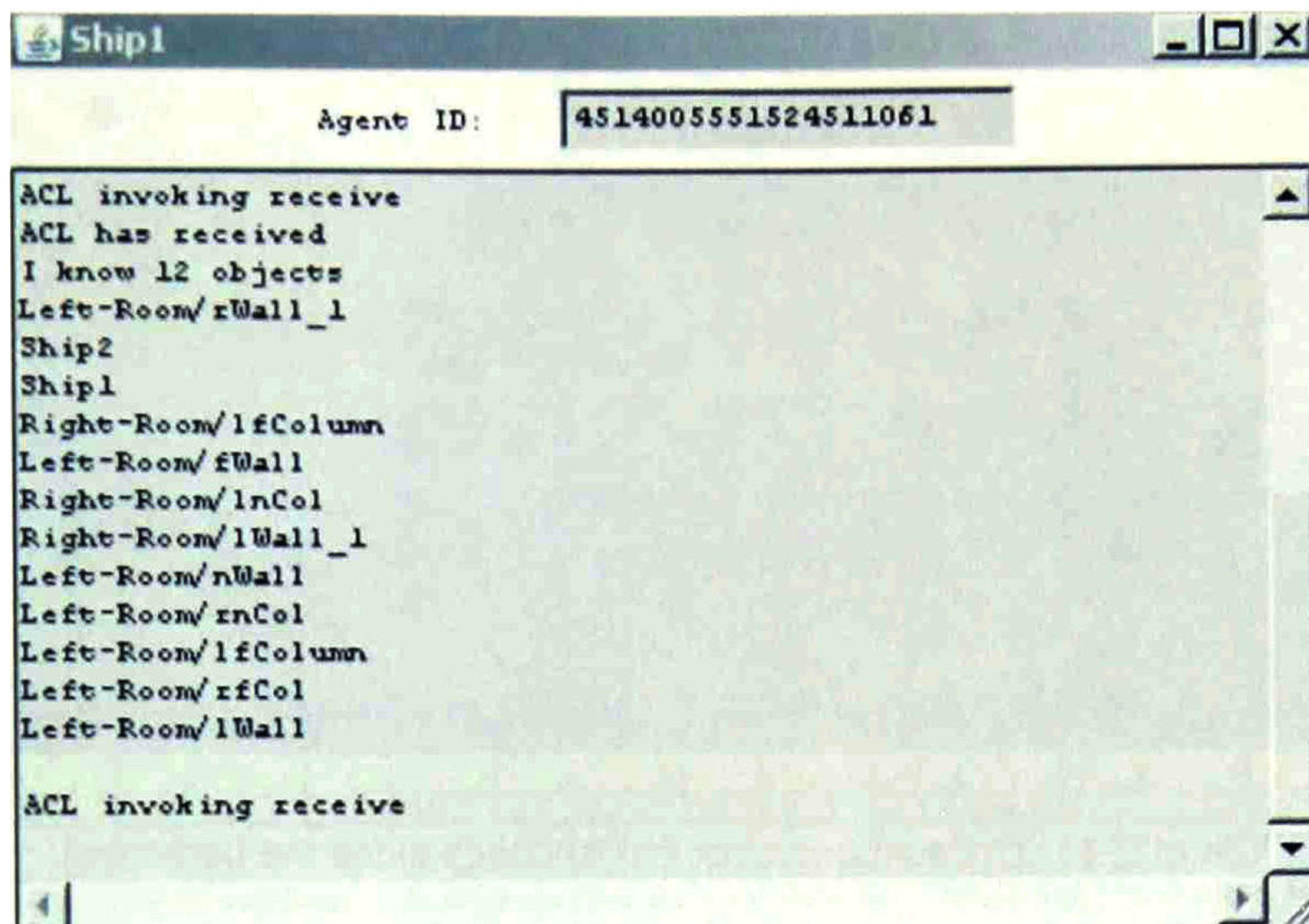


Figure 6-11. Ship1 knows 12 objects of the environment.

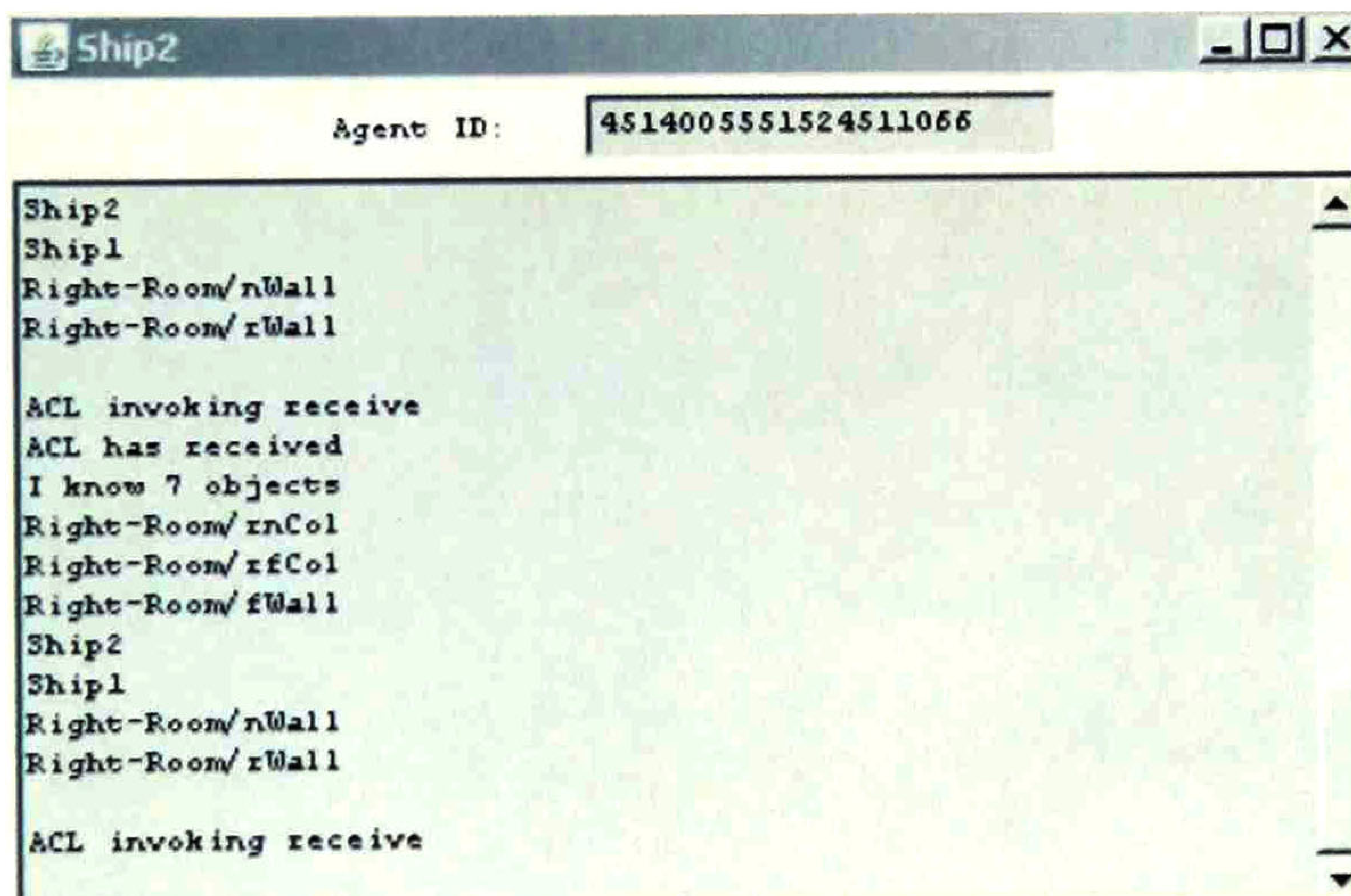


Figure 6-12. Ship2 knows 7 objects of the environment.

The simulation continues as illustrated in Figures 6-13 and 6-14. In Figure 6-14 Ship1 (the Predator) starts its "attack" skill on Ship2. The metrics of the Environment Agent at the same time of the Figure 6-14 are shown in Figure 6-15.

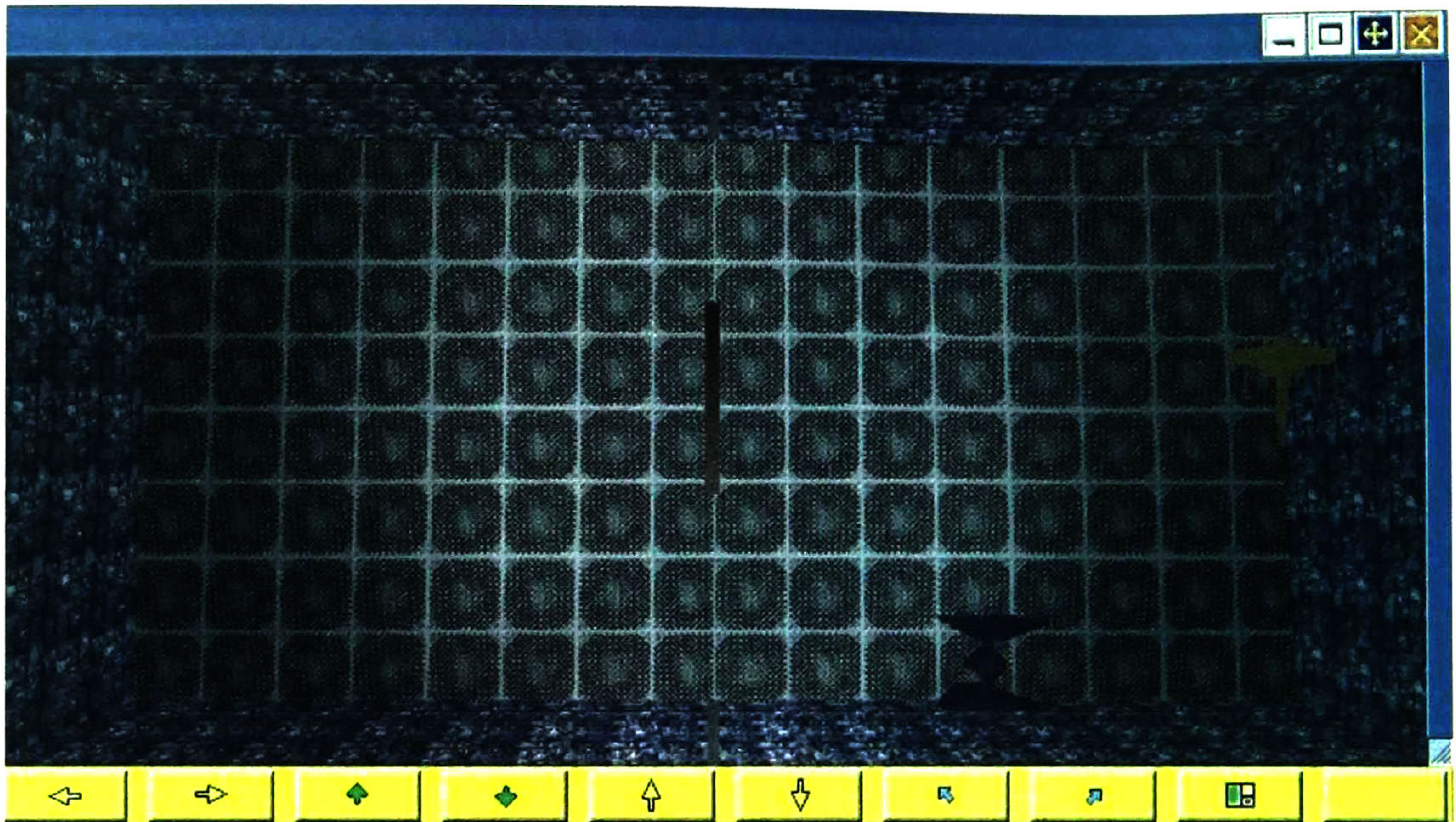


Figure 6-13. The scenario after 437 seconds since the beginning.

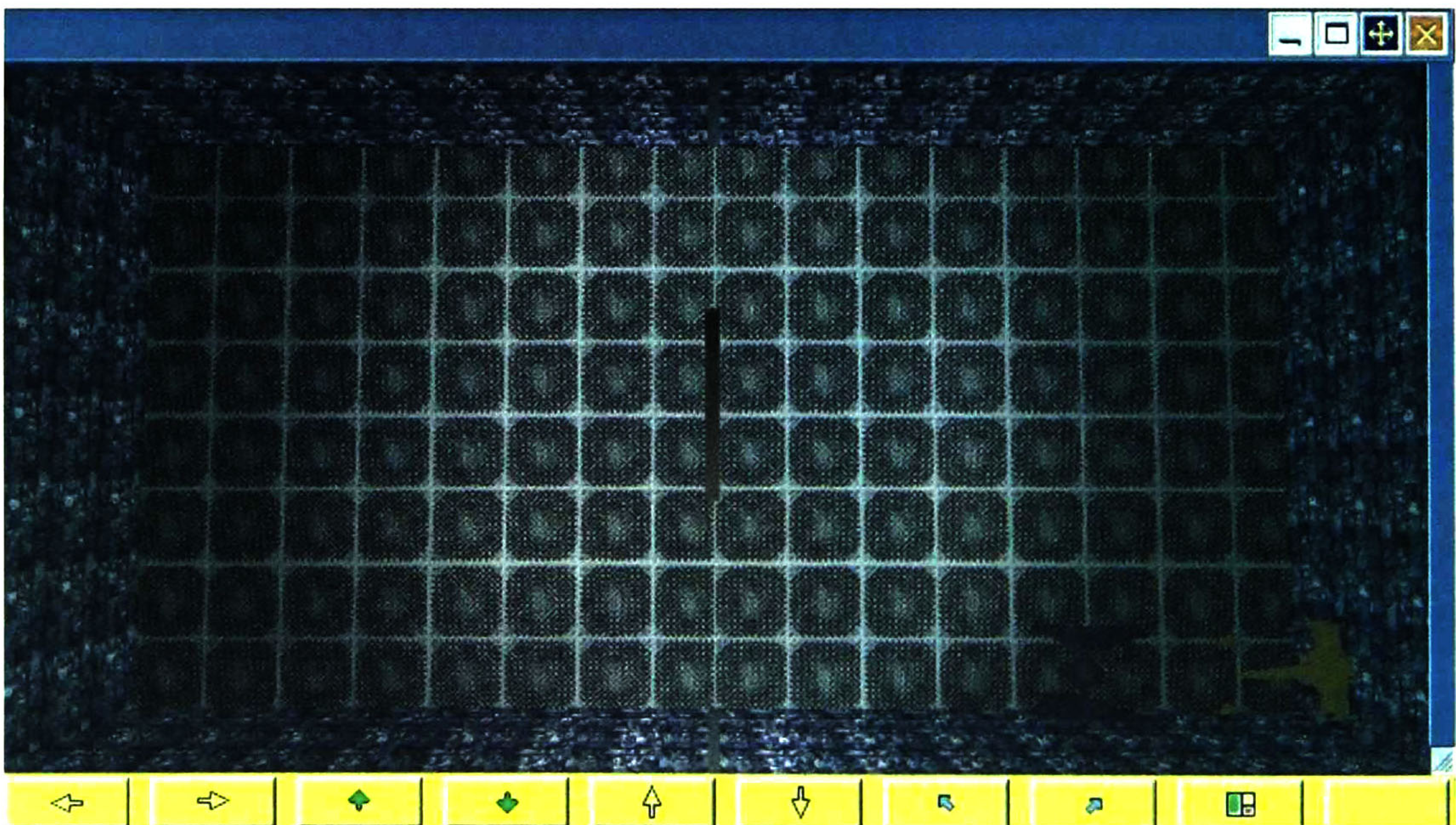


Figure 6-14. The scenario after 478 seconds since the beginning.

The situation reported by the agent that manages Ship1 when reaches the goal "chase" and starting reaching goal "attack" is shown in Figure 6-16. Notice in Figure 6-15 in the box of "object positions" that Ship1 and Ship2 are in the same cube.

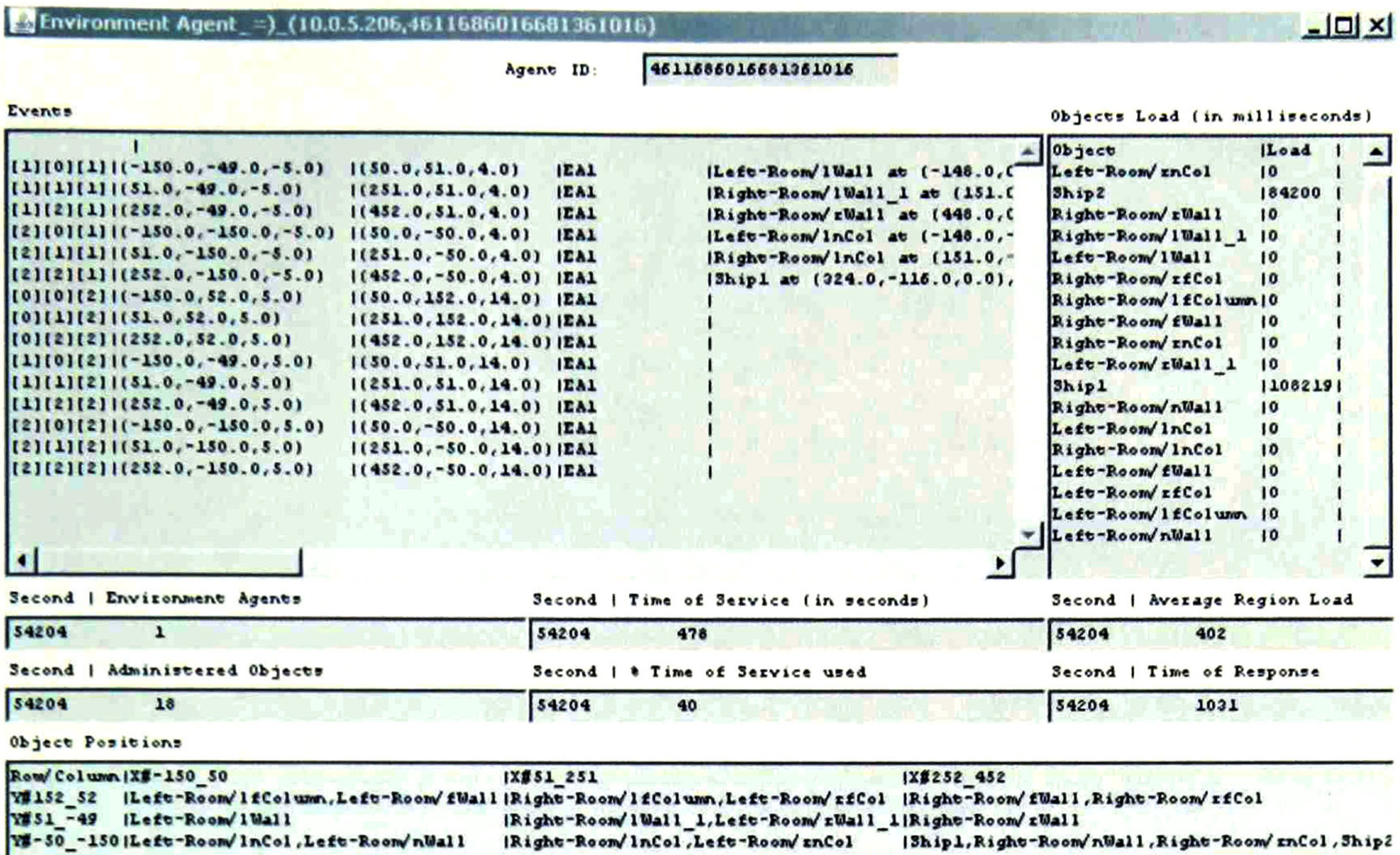


Figure 6-15. The Environment Agent after 478 seconds since the beginning.

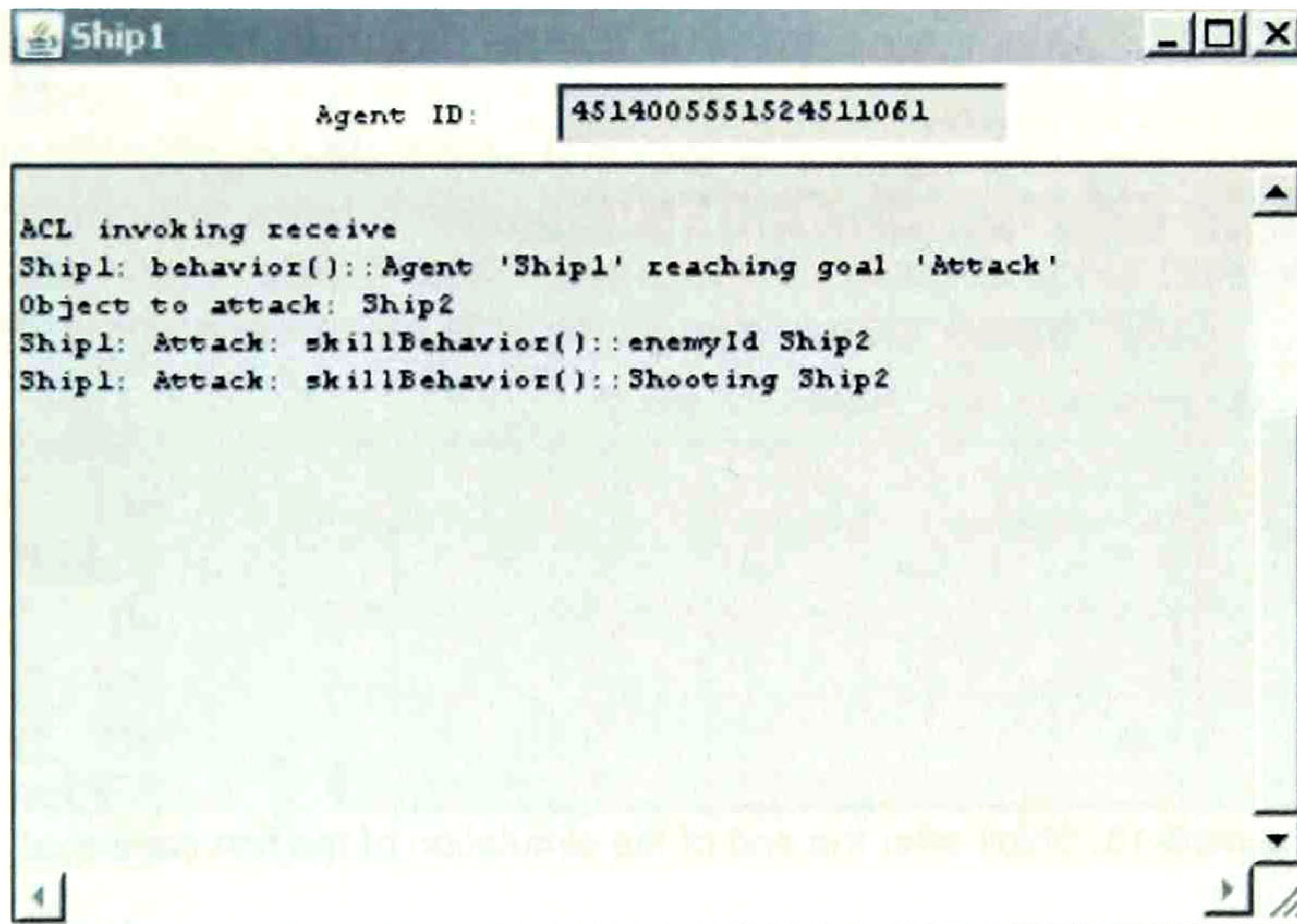


Figure 6-16. Ship1 reaching the goal attack.

The situation after no agent carries out any intention, that is, the end of the simulation is shown in Figures 6-17 and 6-18.

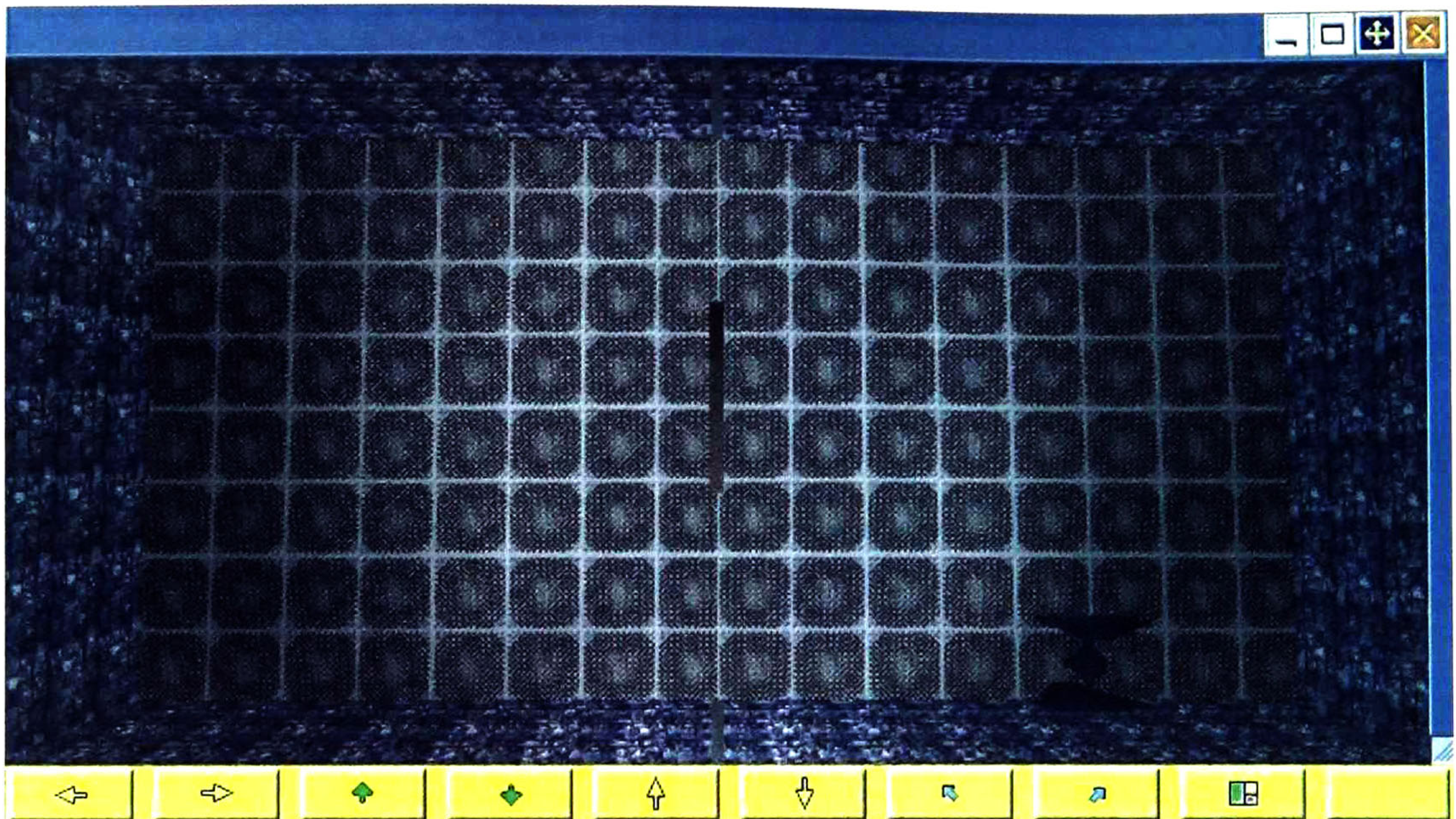


Figure 6-17. The scenario at the end of the simulation of the first case study.

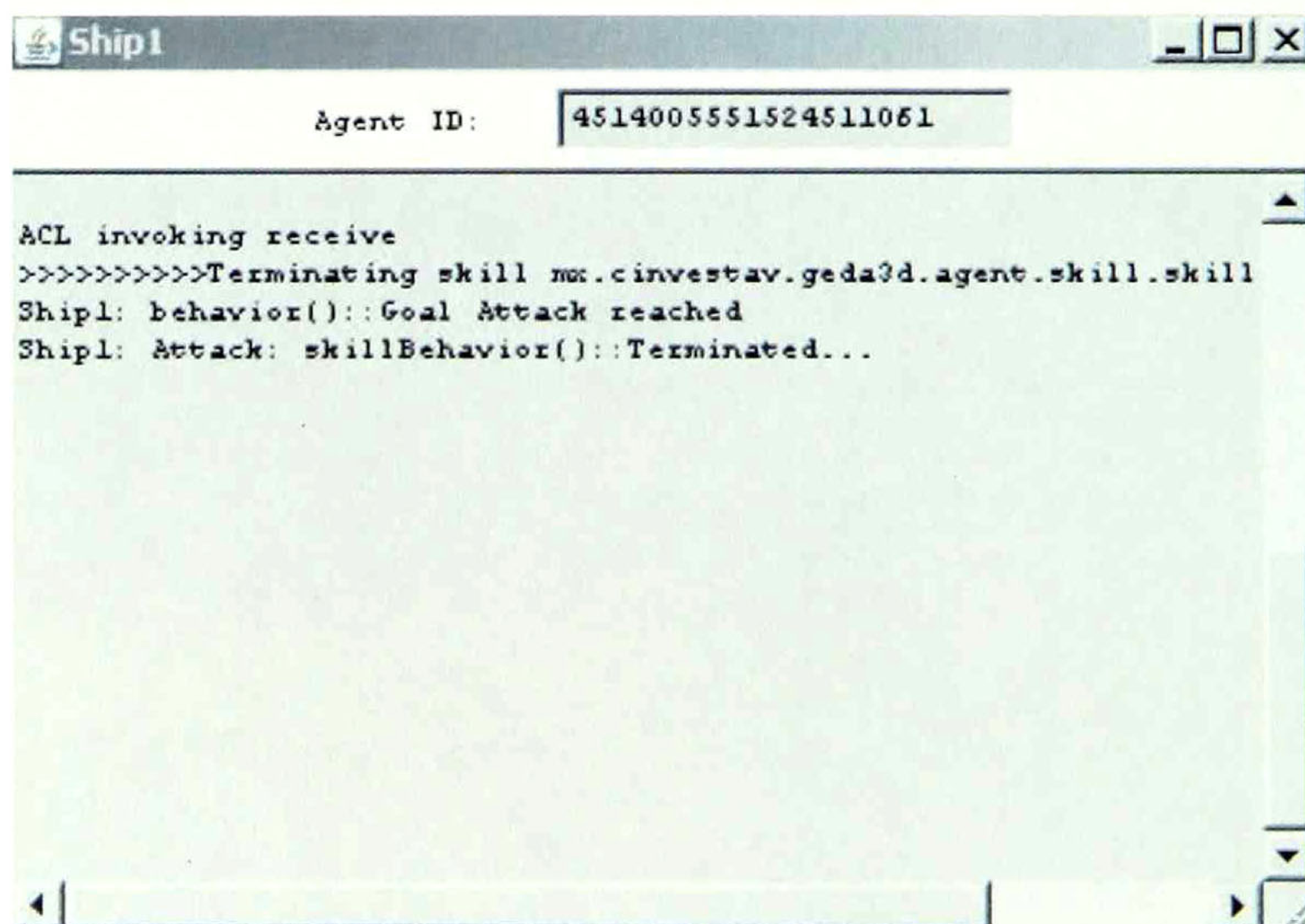


Figure 6-18. Ship1 after the end of the simulation of the first case study.

Finally the Environment Agent shows the environment state excluding Ship2 from the "Objects Load" table and from the "Object Positions"; notice that current metrics decreased in the boxes "Time of service used" and "Average Region Load" The latter situation would lead to a fusion if there were more than one Environment Agents representing the environment (see section 6.2.3).

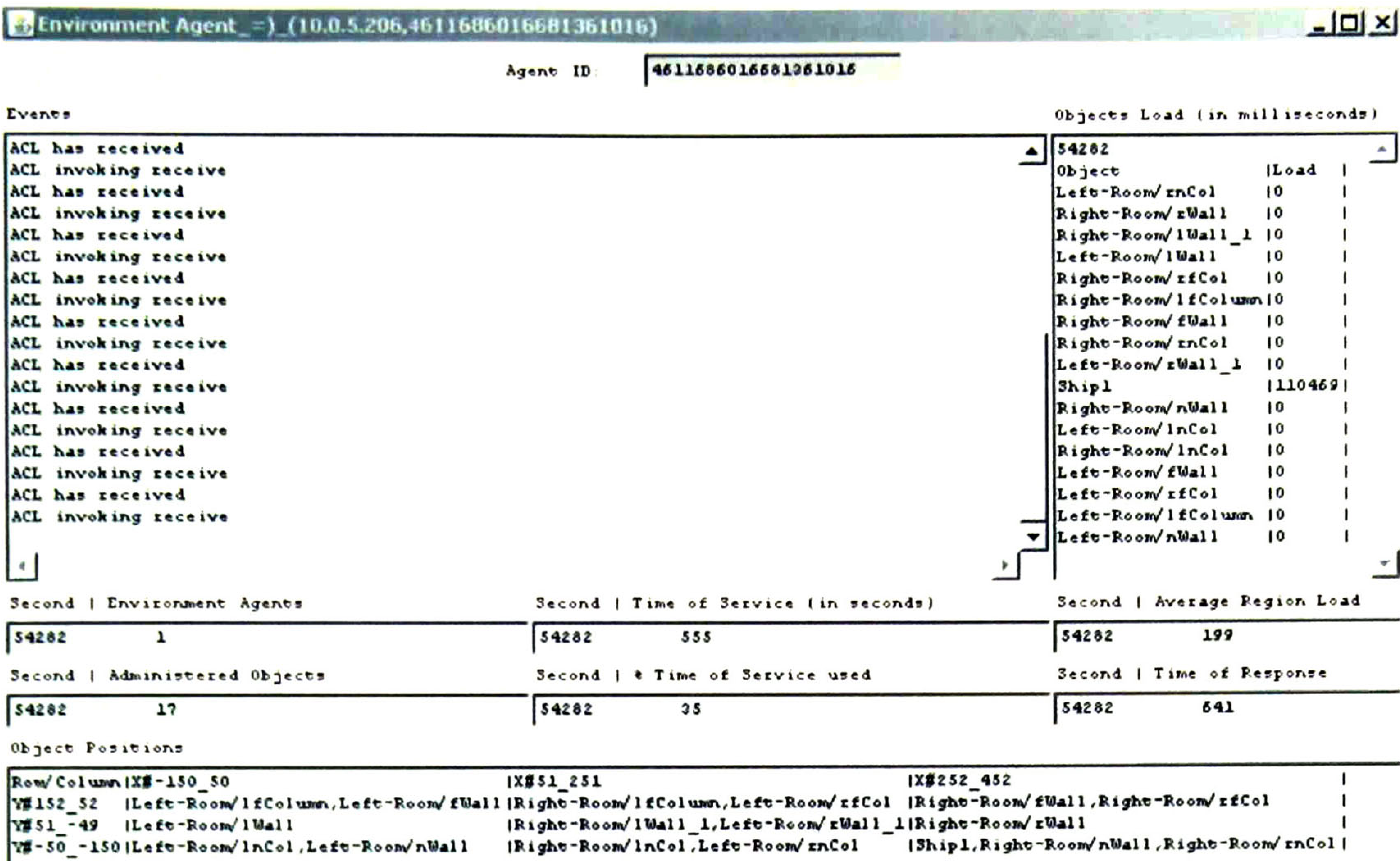


Figure 6-19.The Environment Agent after the end of the simulation of the first case study.

6.2.2 Metrics without using dynamic adaptation policies

This simulation began after 53727 seconds since the beginning of the day (see Figure 6-3) taking the local machine time. This suits well when we use more than one machine and are willing to compare in the same graphic (see section 6.2.3). During this simulation we have only one Environment Agent (EA).

In Figure 6-20 we depict the amount of objects administered by the EA. At the beginning of the simulation the EA administers 18 objects and, after Ship1 attacked Ship2, the latter disappeared of the environment and EA_1 notifies that administers 17 objects.

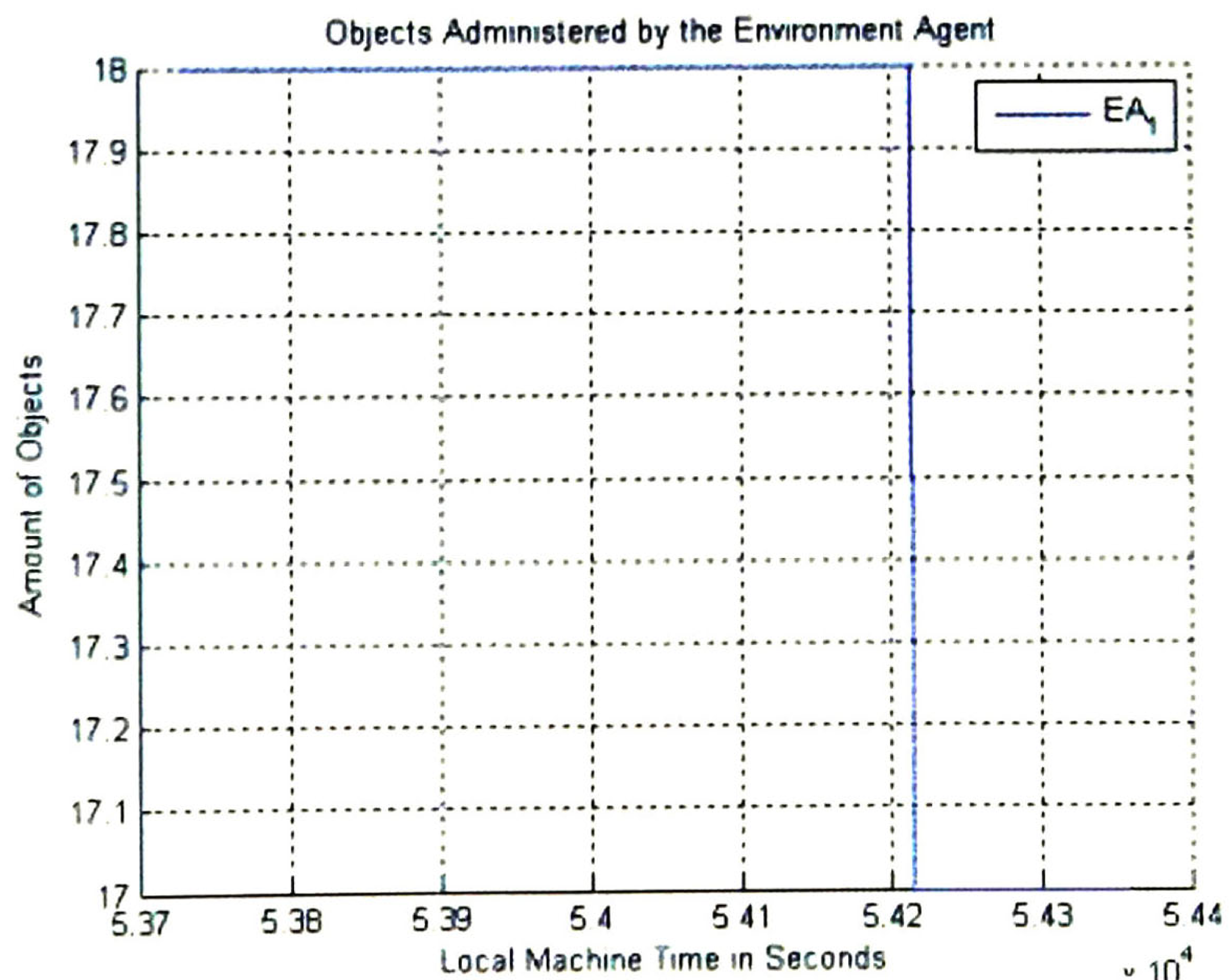


Figure 6-20. Objects Administered using one EA in the first case study.

In Figure 6-21 we depict the average region load of the EA. The load of an object is the amount of *time of service* dedicated to process the intention of an agent that causes modifications to such object.

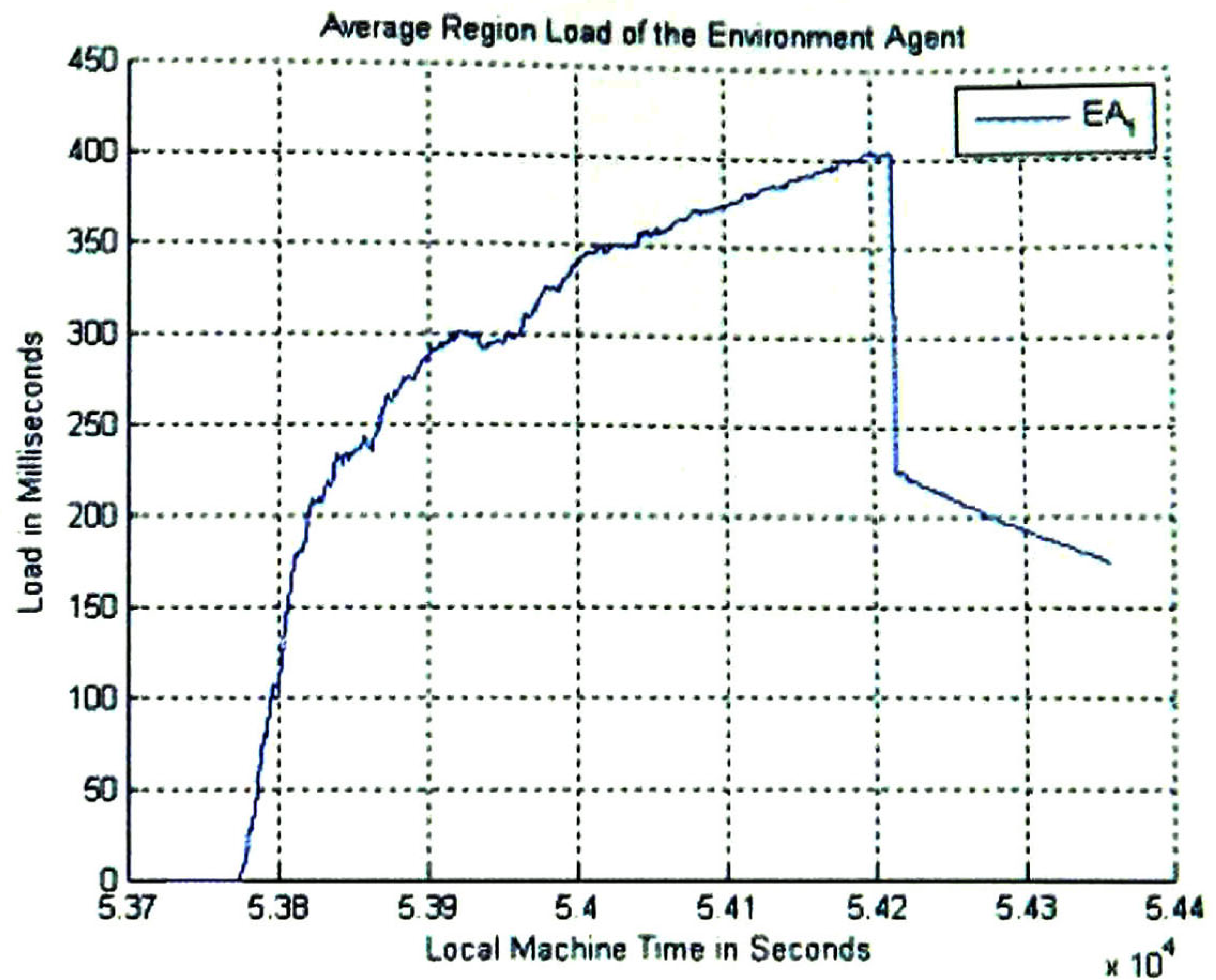


Figure 6-21. Average Region Load using one EA in the first case study.

The load of the *region* administered by the EA is the sum of the accumulated loads of all objects that are either members or neighbors of the *region* (see section 5.4). After an object disappeared of the environment, the average region load does not count the removed object´s load, as if such object never existed.

In Figure 6-22 we depict the *time of response* for the processed intentions, such time is counted since intention arrived to MailBox until the EA finished processing the intention. If such time increases considerably, this would cause the starting of a dichotomy process (see section 6.2.3).

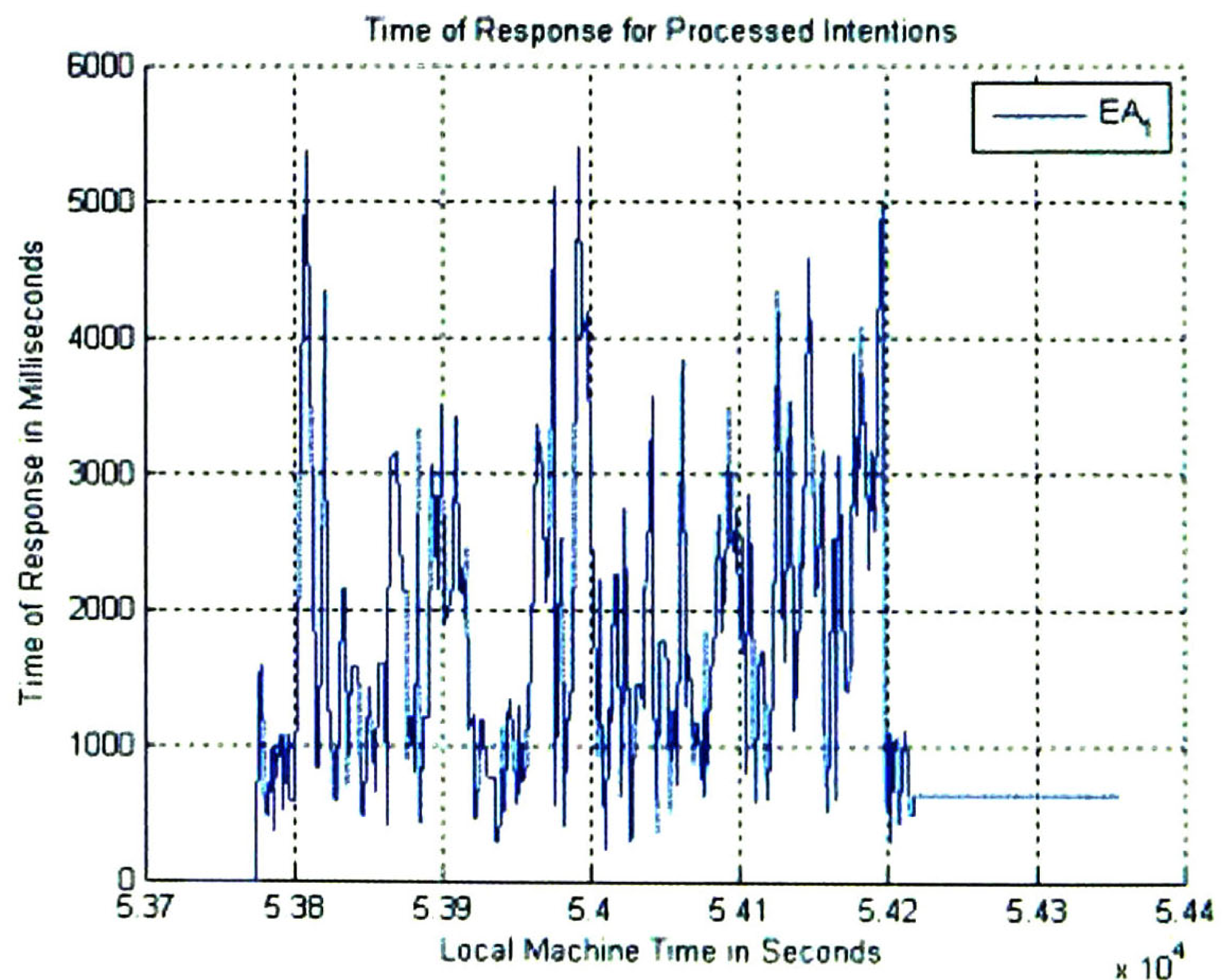


Figure 6-22. Time of Response using one EA in the first case study.

In Figure 6-23 we depict the percentage of *time of service* that the EA dedicated to process agent intentions. After Ship1 reached its "attack goal", the latter and Ship2 did not send intentions to the Environment Agent anymore and, its percentage of *time of service* decreased.

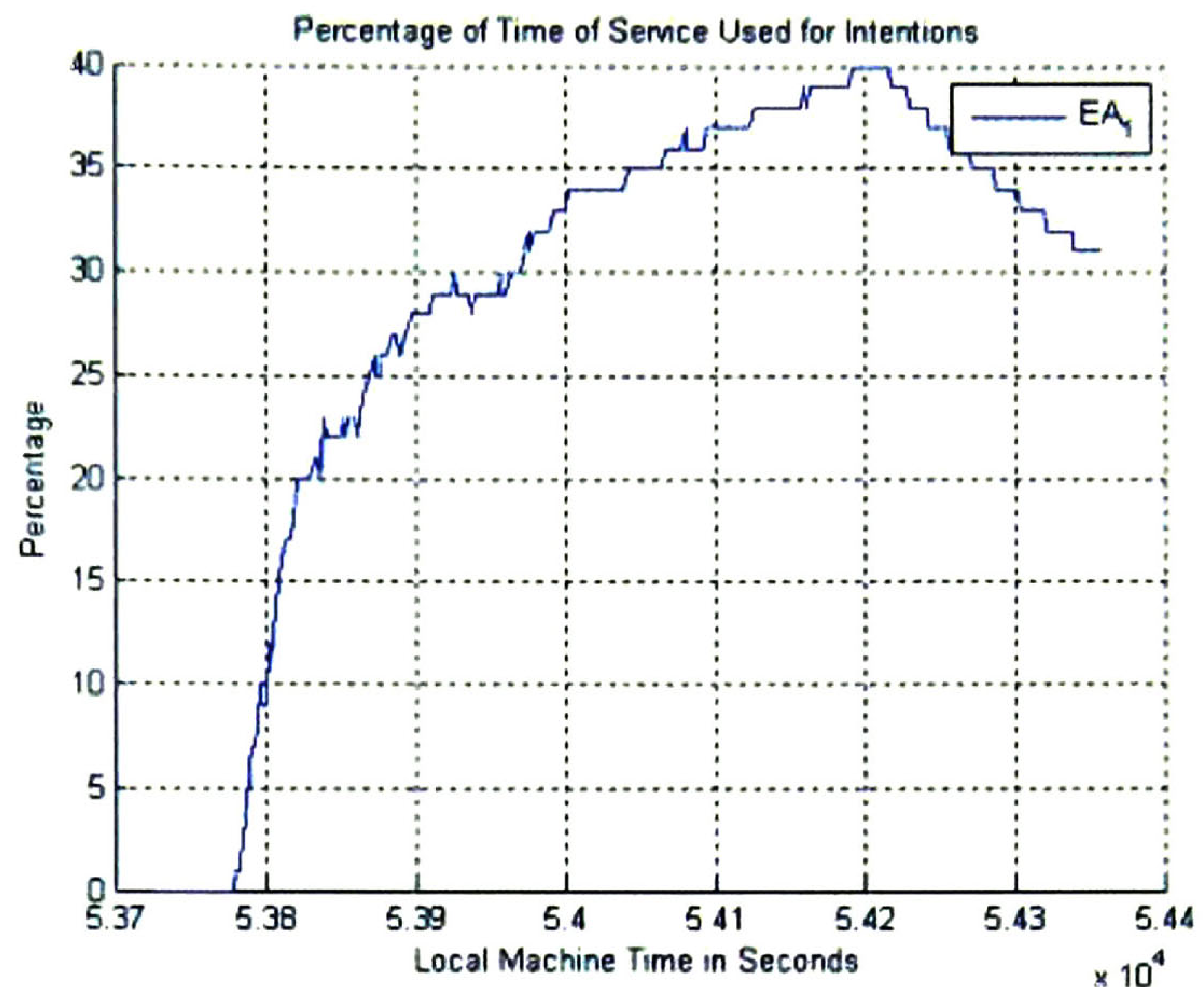


Figure 6-23 Percentage of Time of Service using one EA in the first case study.

6.2.3 Metrics applying dynamic adaptation policies

In Figure 6-24 we depict the amount of agents running during this simulation. Notice that EA_1 shows a delay in notifying that it has applied a dichotomy process when there are two Environment Agents. After approximately 60 seconds a fusion process is started and EA_1 sends its information to EA_2 which continues attending the entire environment.

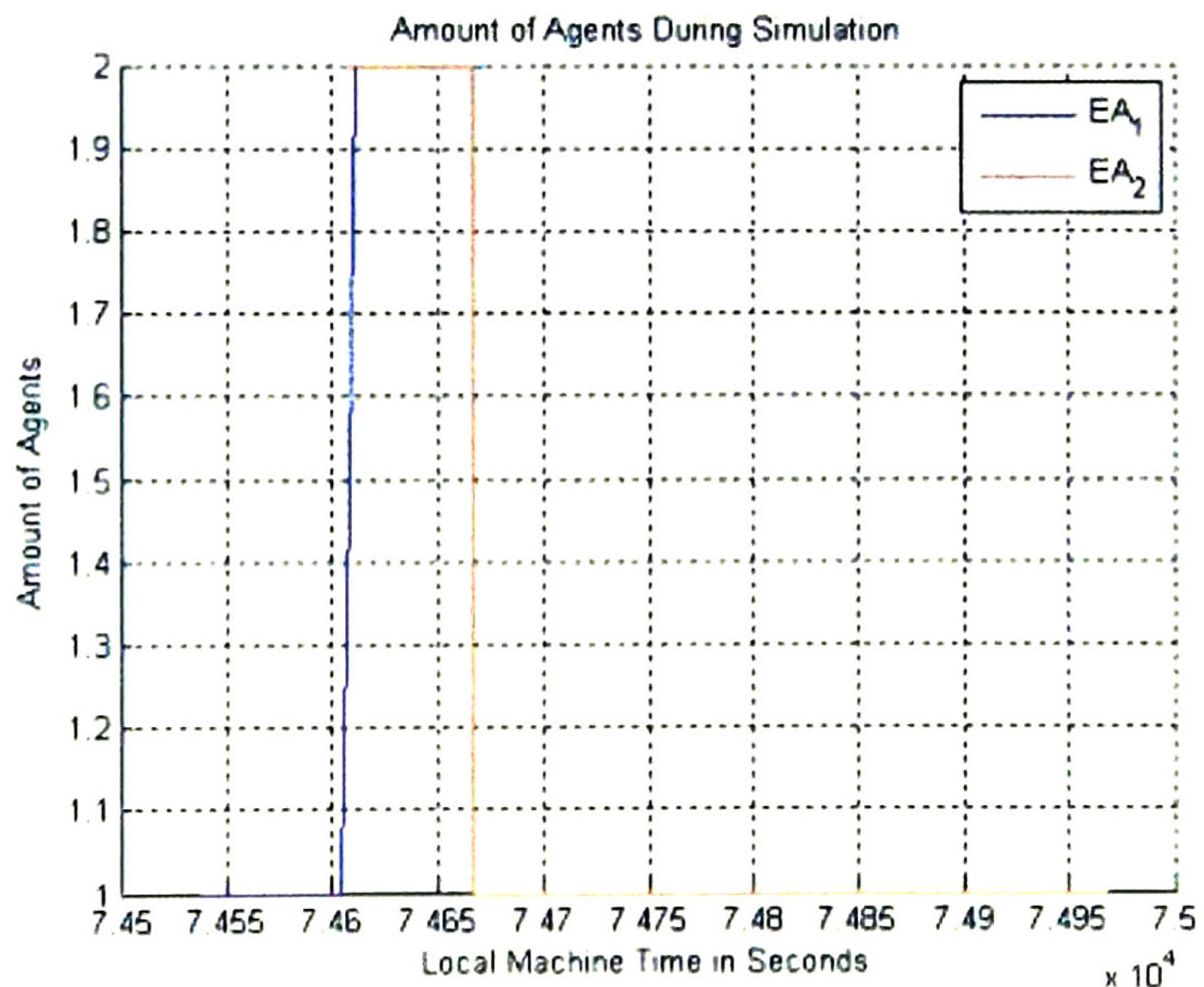


Figure 6-24. Amount of agents using up to two EAs in the first case study.

In Figure 6-25 we depict the amount of objects administered by the EAs. At the beginning of the simulation the EA_1 administers 18 objects. After the dichotomy process, EA_1 administers 2 objects and EA_2 administers 16 objects. At every moment the sum of all objects administered by the two EAs is 18.

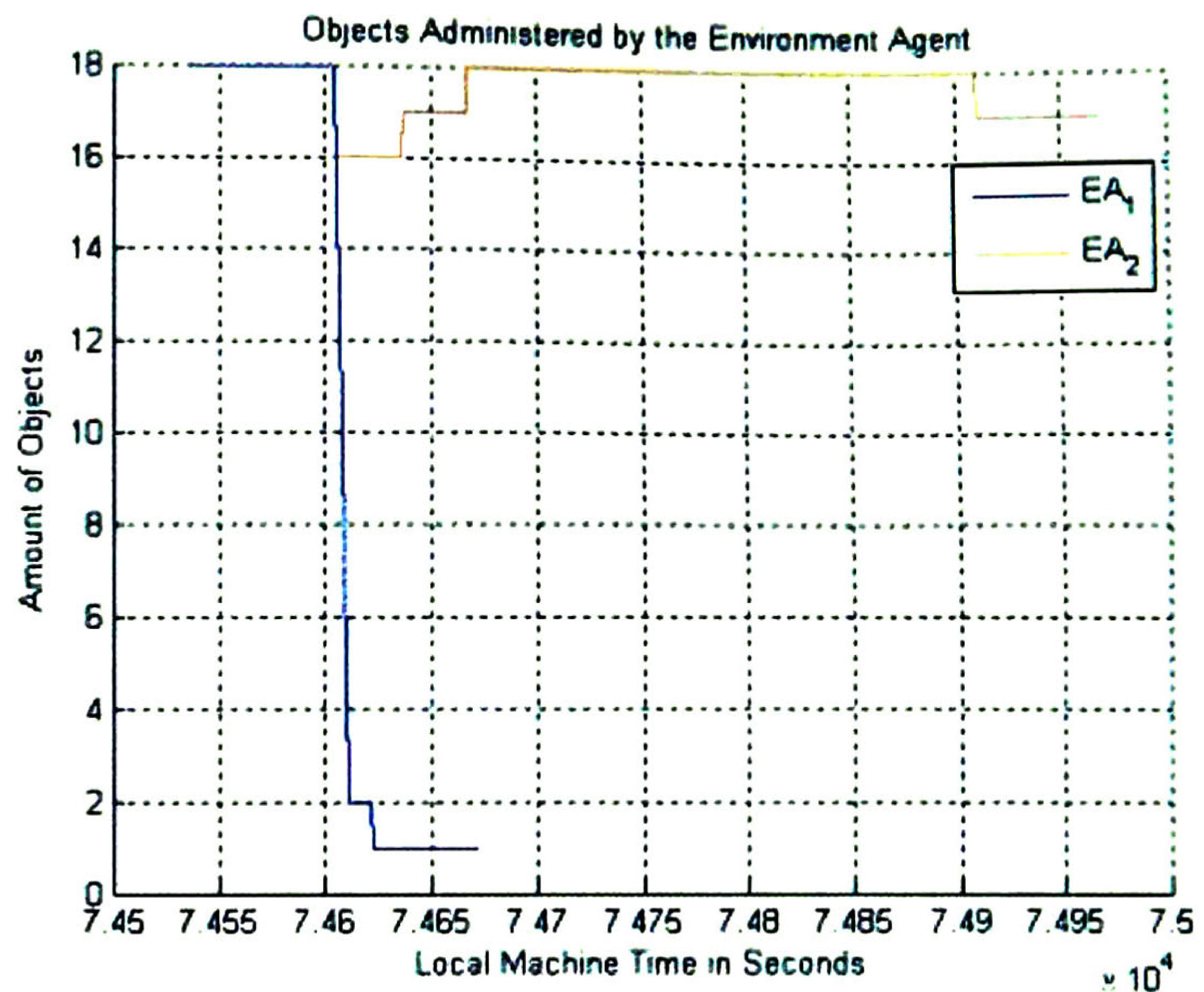


Figure 6-25. Objects Administered using up to two EAs in the first case study.

In the previous Figure 6-25, after some elapsed time, EA_1 reports that it administers only one Environment Object and EA_2 administers 17 objects. This situation happened because at time 74623 the object Ship1 moved from the region administered by EA_1 to the region of EA_2. Sometime after 74623, EA_2 stopped notifying EA_1 about Ship1 movements because no part of Ship1 was occupying the region of EA_1. After time 74667 EA_1 carried out a fusion process with EA_2. After Ship1 attacked Ship2, the latter disappeared of the environment at time 74911.

In Figure 6-26 we depict the average region load of the EAs. Comparing this Figure with 6-21 is notable that the load of around 400 milliseconds was shared among the two Environment Agents, being the maximum around 200 milliseconds.

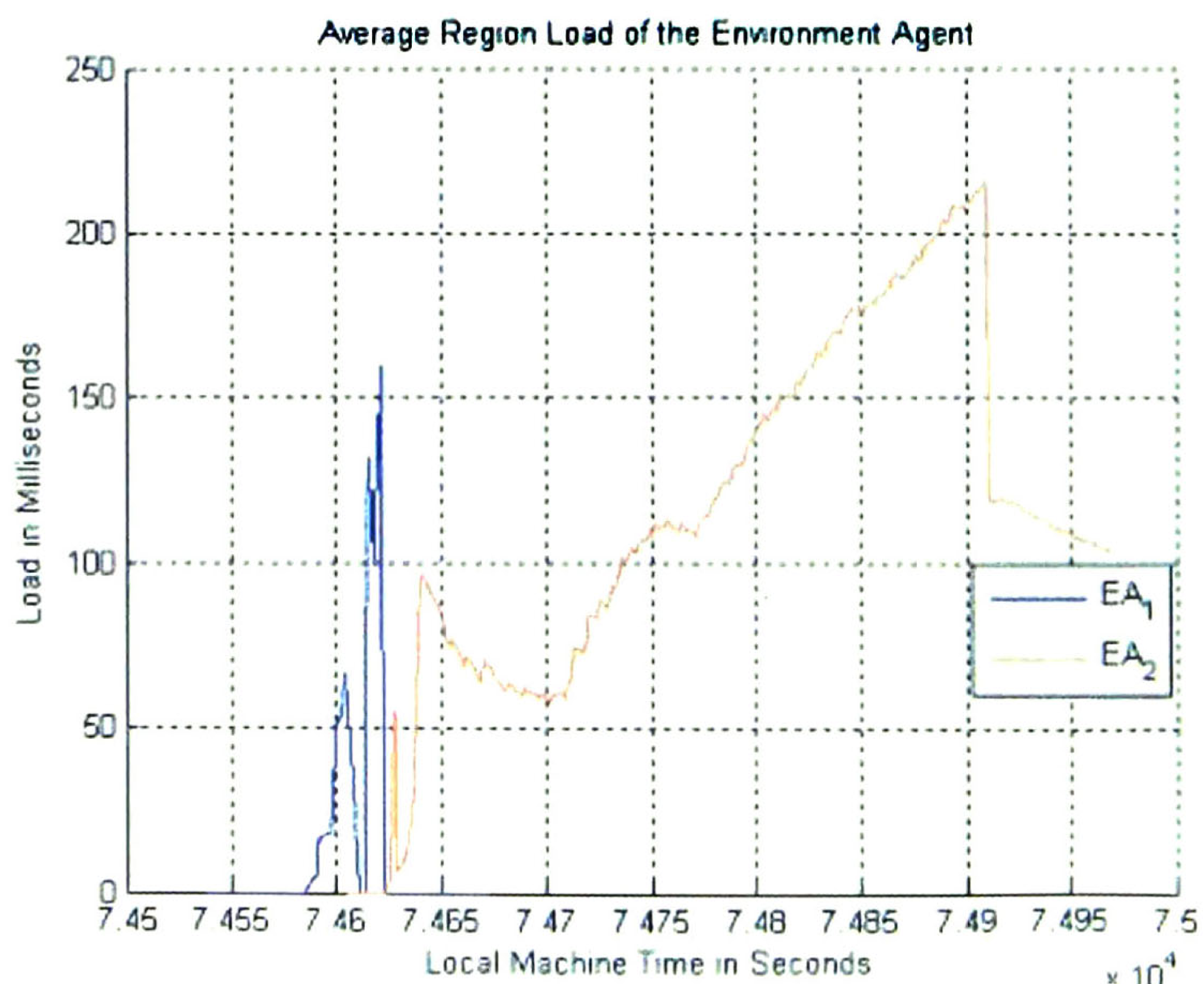


Figure 6-26. Average Region Load using up to two EAs in the first case study.

In Figure 6-27 we depict the *time of response* for the processed intentions of the EAs. At time 74613 such time increased three consecutive times and, caused the starting of a dichotomy process (see also previous figures 6-24, 6-25 and 6-26). This figure shows a pending problem to be solved.

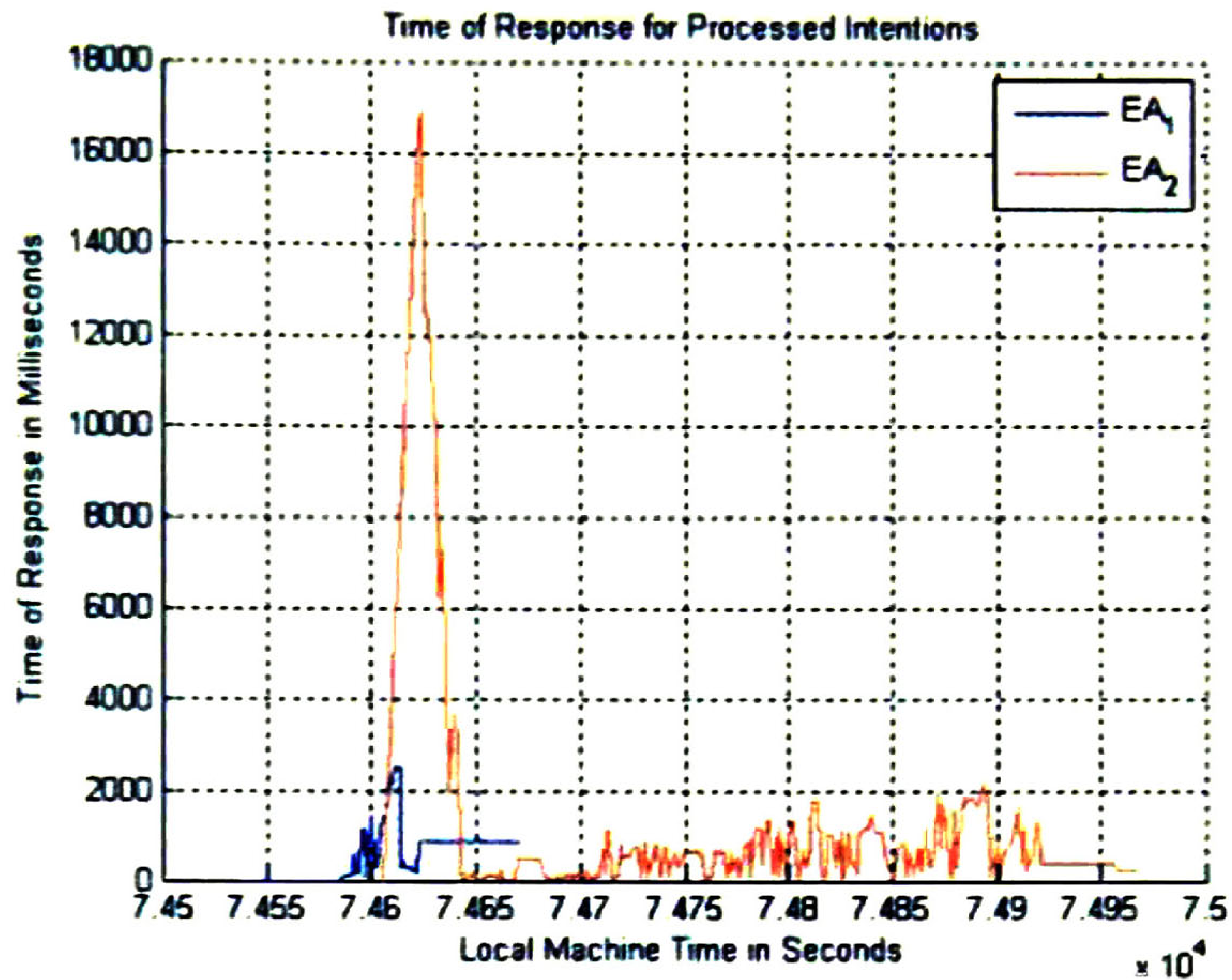


Figure 6-27. Time of Response using up to two EAs in the first case study.

In the previous Figure 6-27 the setting up of a new EA, until it is ready to process the first intention, delays a considerable time due to two factors: messages from this agent are at the moment treated as ordinary messages, that is, its messages have to wait until other queued processes´ messages are sent; the same happens with group management issues requested by the second EA, the CriticalCoordinator process is used for mutual exclusion during group creations and group joins and, its messages are also treated as ordinary (see section 7.2).

In Figure 6-28 we depict the percentage of *time of service* of the EAs. After Ship1 moved from Region 1 to Region 2 the percentage of EA_1 decreased causing a fusion process. After Ship1 reached its "attack goal", the latter and Ship2 did not send intentions to EA_2 and, its percentage of *time of service* also decreased.

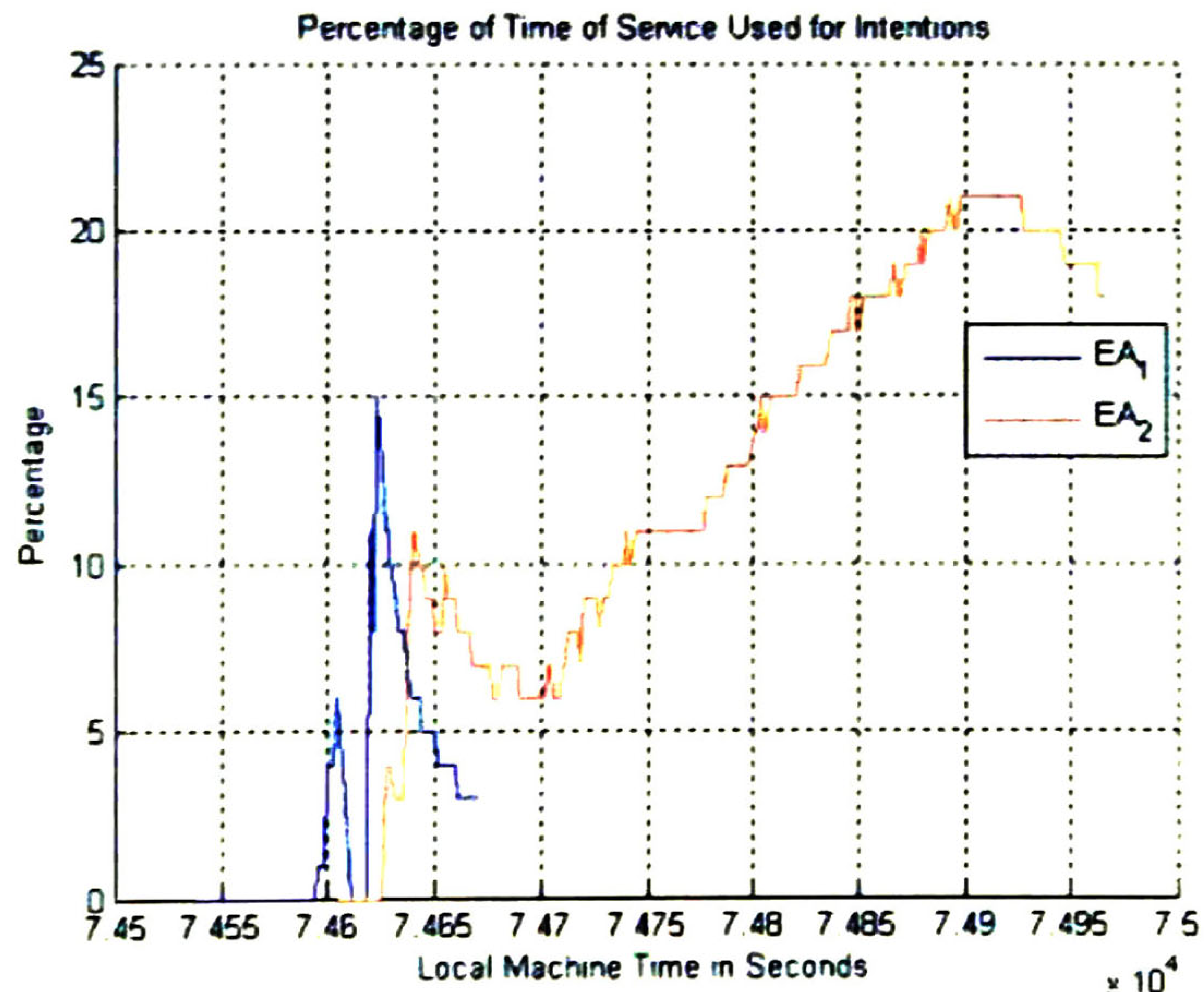


Figure 6-28 Percentage of Time of Service using up to two EAs in the first case study.

Comparing the previous Figure 6-28 with 6-23, it is also notable that the percentage of *time of service* dedicated by these two Environment Agents was divided almost to half.

6.3 Prey Predator Avatar Chasing

We present here the case study of the work developed by [MARTINEZ] for the GeDA-3D Rendering using our Middleware. This Rendering is in charge of receiving and interpreting lower level commands that receives in XML format in order to display the graphical state of an avatar or an object in the environment.

6.3.1 Illustrating the case without using dynamic adaptation policies

In the first case study both Rendering and Scene Descriptor were written as a single high coupled application displaying the scenario after interpreting the scene and scenario script. The second case of study begins using an agent called Scenario Descriptor Emulator (SDE) which in fact emulates the out of the Scenario Descriptor in order to avoid using such program that at the moment of these tests was still under construction. An advance in the deployment of the Scenario Descriptor is exposed in section 6.4.

When SDE runs, it sends an XML description to the Scene Descriptor V2 illustrated in Figure 6-29. The latter is a prototype of the interface to be provided to solely write the script for the scene, separated from the interface dedicated to the scenario description (see section 3.4.2).

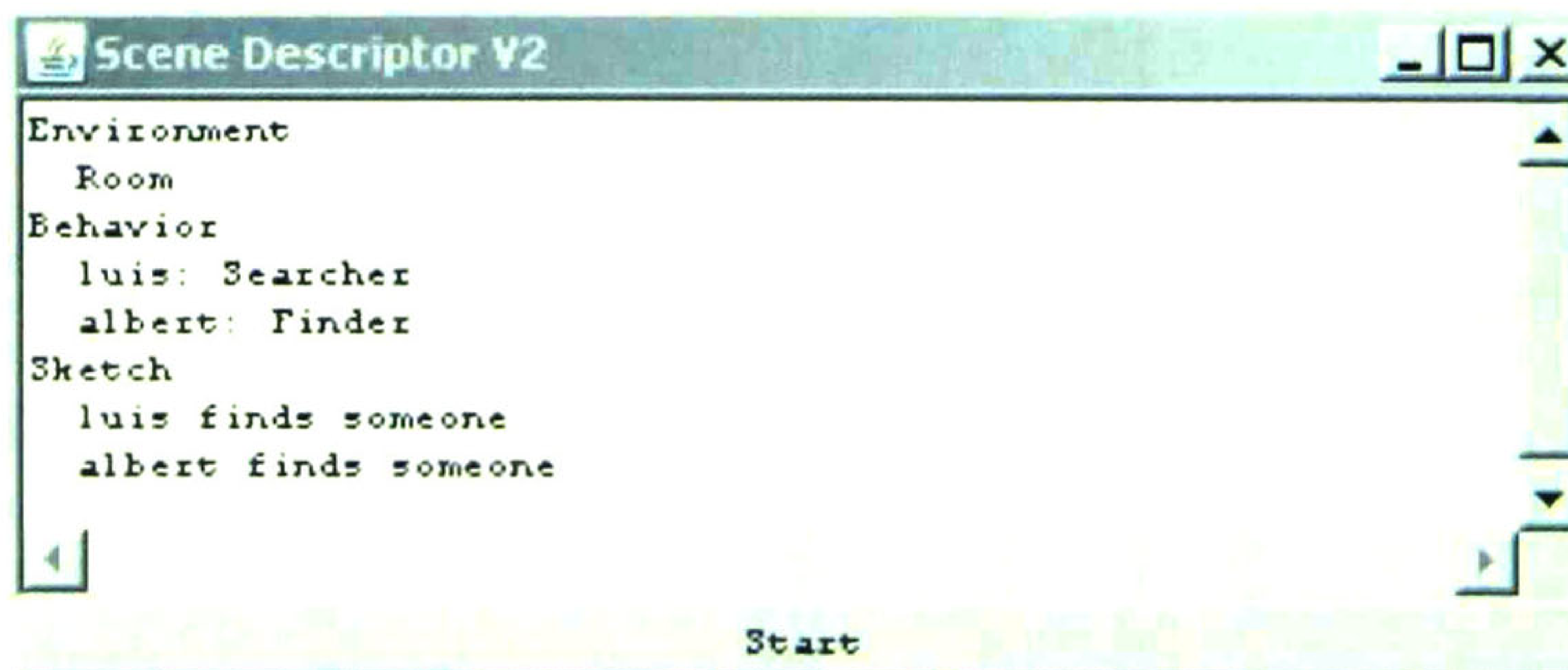


Figure 6-29. The prototype of the Scene Descriptor version 2 for the second case study.

When the user presses the start button, this Scene Descriptor requests the platform to launch the Environment Agent (EA) for this environment and, the agents luis of class Searcher and albert of class Finder. After the EA is launched, SDE sends to the EA the scenario description in XML format.

When the EA receives the scenario description sends it to the Rendering, loads the environment classes designed to the current environment and, shows the

amount of running "Environment Agents", the amount of "administered objects" each "objects load" the metrics for "time of service", "percentage of time of service used", "average region load" and the latest "time of response" and at the bottom it shows all "objects positions" in the virtual environment; see Figures 6-30 and 6-31.

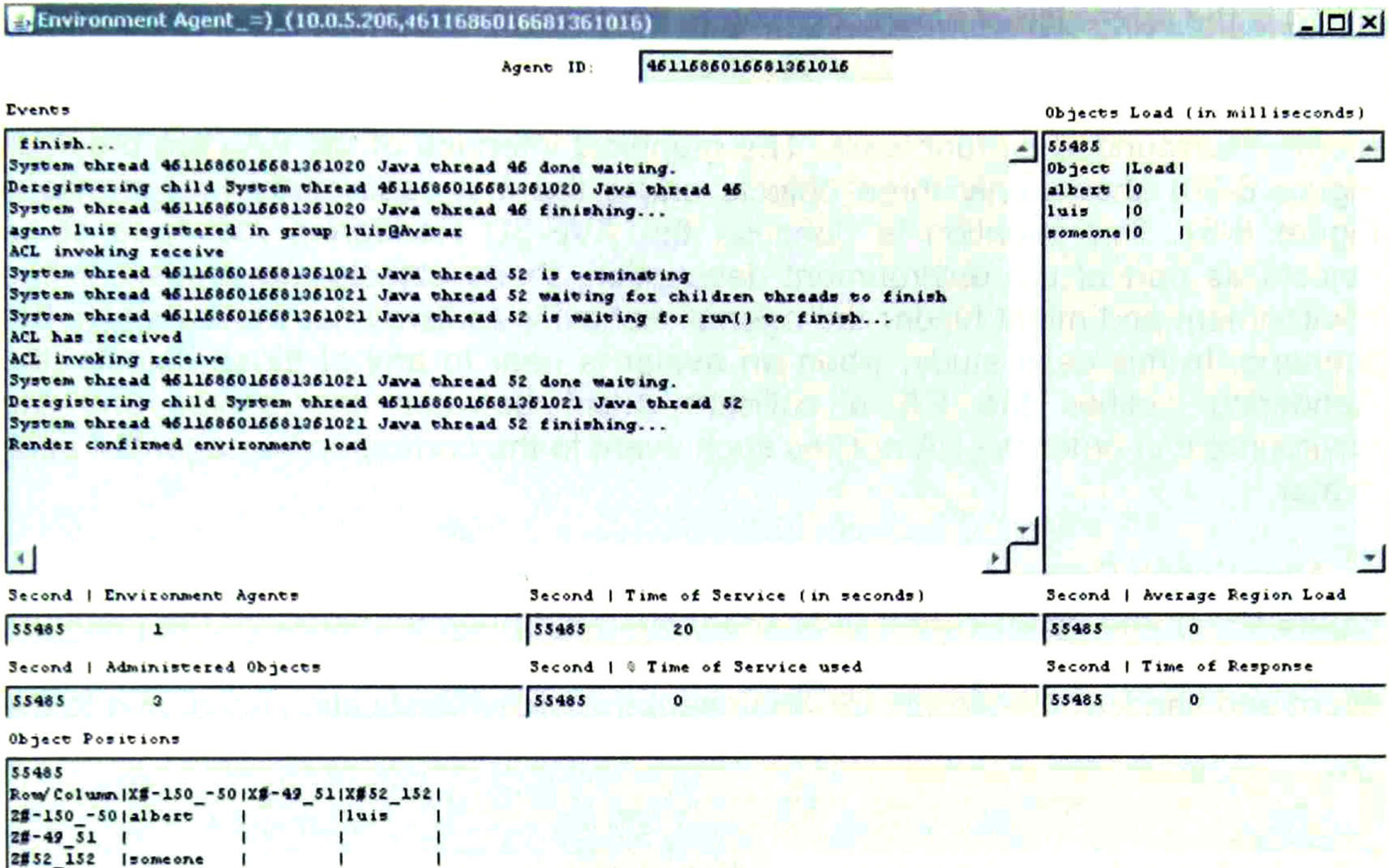


Figure 6-30. The Environment Agent at the beginning of the first case study.

Figure 6-31. The Scenario´s state: at the beginning of the second case study.

For previous Figure 6-30, "Object positions" shows the content of 9 of 27 Environment Cubes, using as rows the coordinate *z* and as columns the coordinate *x* because the EA determined that this environment was a two dimensional arrangement having all objects in the same coordinate *y*. The center of the scenario is at (x=0, y=0, z=0). All boxes of the EAs interface show a label "second" which is the reference of time according to the local machine

In previous Figure 6-31 six boxes appear as part of the environment and this room is surrounded by four walls. The graphical interface of the EA (see previous Figure 6-30) shows only three objects unlike the first case study (see previous Figure 6-3). This situation is because the AVE-3D Rendering manages some objects as part of the environment description; these objects are present in the environment and might hinder the agents' behavior as its avatar moves inside the scenario. In this case study, when an avatar is near to any of those objects, the Rendering notifies the EA a collision event between the avatar and the environment in order the EA notifies such event to the corresponding agent of such avatar.

After Scene Descriptor V2 requested the platform to launch agents for luis (see Figure 6-32) and albert (see Figure 6-33) such agents are loaded on the platform. After each agent is loaded on the platform, the Scene Descriptor is notified of such event and sends to the corresponding agent the goal specification according to the sketch (see previous Figure 6-29 and Figures 6-32 and 6-33).

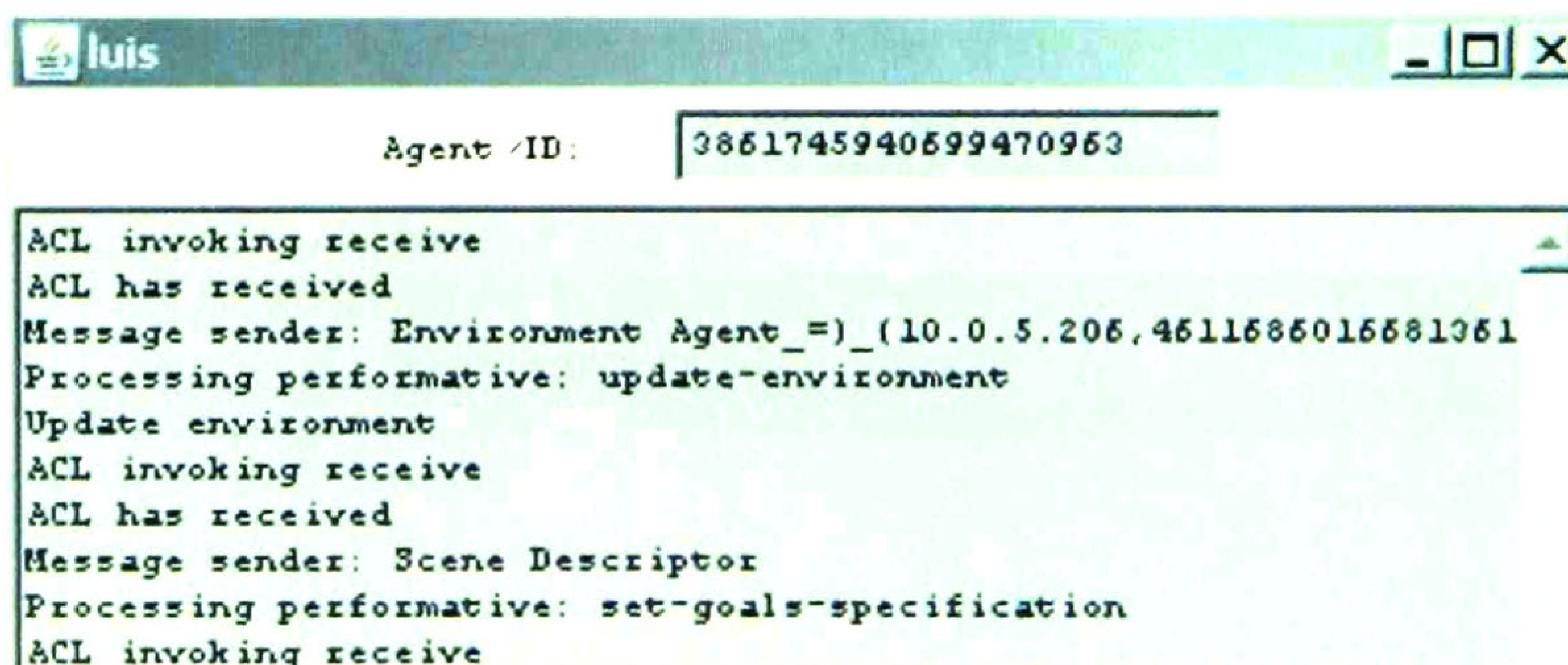


Figure 6-32. Agent luis receives its goal specification.

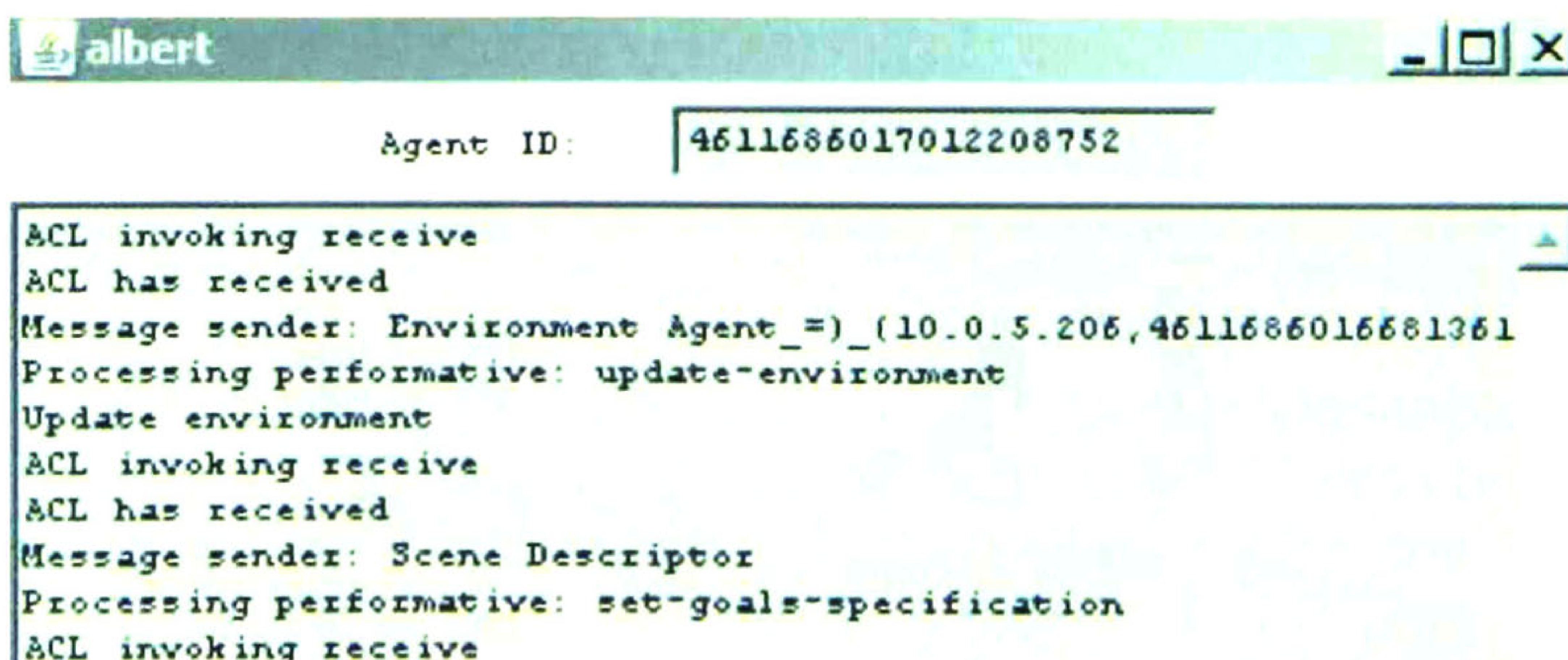


Figure 6-33. Agent albert receives its goal specification.

All graphical interfaces but the Rendering and the Scene Descriptor are optional to be shown.

After the EA loaded the environment classes, it request platform to join agents luis and albert to a group created for every object near to their avatars (including the group created for their own avatars). The EA sends to all agents only the objects that are around them. After both agents are ready, they send to the EA their intentions to move their avatars; see Figures 6-34 and 6-35. Agent luis controls the white skin male avatar and, agent albert controls the 2nd male avatar.

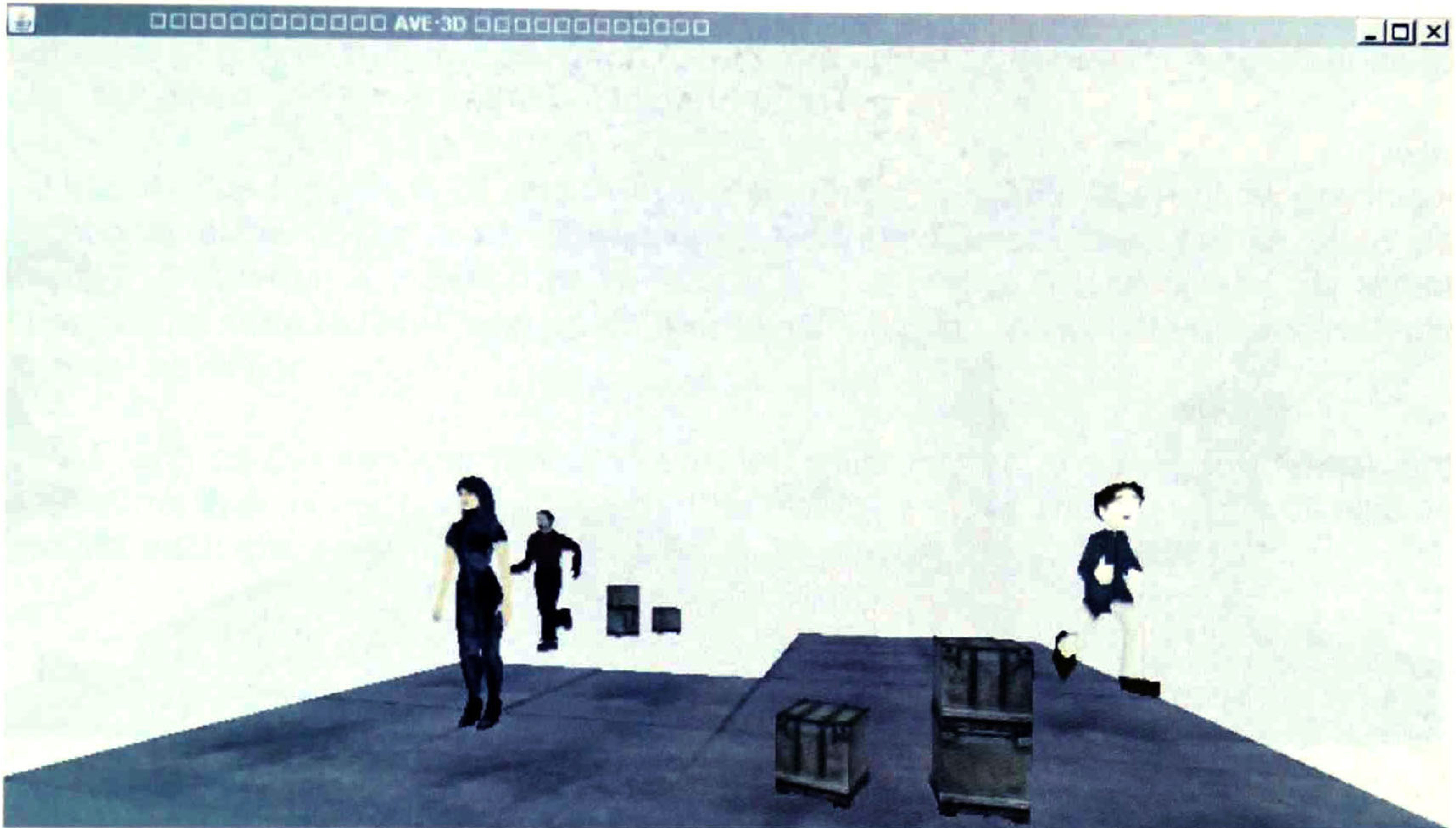


Figure 6-34. The scenario after 40 seconds since the beginning.

Environment Agent_=)_(10.0.5.206,4611686016681361016)

Agent ID: 4611686016681361016

Events

```
0,51.0,152.0) |EA1          |                              |
[2][0][2]|(-150.0,-150.0,52.0)  |(-50.0,-50.0,152.0)|EA1     |
[2][1][2]|(-49.0,-150.0,52.0)   |(51.0,-50.0,152.0) |EA1     |
[2][2][2]|(52.0,-150.0,52.0)    |(152.0,-50.0,152.0)|EA1     |

ACL has received
ACL invoking receive
```

Objects Load (in milliseconds)

```
55504
Object  |Load|
albert  |1406|
luis    |1436|
someone |0   |
```

Second | Environment Agents: 55505 1

Second | Time of Service (in seconds): 55505 40

Second | Average Region Load: 55504 74

Second | Administered Objects: 55505 3

Second | % Time of Service used: 55504 5

Second | Time of Response: 55504 734

Object Positions

```
55504
Row/Column|X#-150_-50|X#-49_51|X#52_152|
Z#-150_-50|          |        |        |
Z#-49_51  |albert    |        |luis    |
Z#52_152  |someone   |        |        |
```

Figure 6-35. The Environment Agent after 40 seconds since the beginning.

In Figure 6-35, the graphical interface of the EA shows in the Object Positions area the current locations of avatars luis, albert and someone (the female). Another update is shown in Figures 6-36 and 6-37.

Figure 6-36. The scenario after 75 seconds since the beginning.

```
Environment Agent_=)_(10.0.5.206,4611686016681361016)

Agent ID: 4611686016681361016

Events
.0,51.0) |EA1          |
[1][0][1]|(-150.0,-49.0,-49.0)  |(-50.0,51.0,51.0)   |EA1   |
[1][1][1]|(-49.0,-49.0,-49.0)   |(51.0,51.0,51.0)    |EA1   |
[1][2][1]|(52.0,-49.0,-49.0)    |(152.0,51.0,51.0)   |EA1   |
[2][0][1]|(-150.0,-150.0,-49.0) |(-50.0,-50.0,51.0)  |EA1   |
[2][1][1]|(-49.0,-150.0,-49.0)  |(51.0,-50.0,51.0)   |EA1   |
[2][2][1]|(52.0,-150.0,-49.0)   |(152.0,-50.0,51.0)  |EA1   |
[0][0][2]|(-150.0,52.0,52.0)    |(-50.0,152.0,152.0) |EA1   |
[0][1][2]|(-49.0,52.0,52.0)     |(51.0,152.0,152.0)  |EA1   |
[0][2][2]|(52.0,52.0,52.0)      |(152.0,152.0,152.0) |EA1   |
[1][0][2]|(-150.0,-49.0,52.0)   |(-50.0,51.0,152.0)  |EA1   |someone at (-60.0,0.0,90.
[1][1][2]|(-49.0,-49.0,52.0)    |(51.0,51.0,152.0)   |EA1   |
[1][2][2]|(52.0,-49.0,52.0)     |(152.0,51.0,152.0)  |EA1   |luis at (140.0,0.0,125.49
[2][0][2]|(-150.0,-150.0,52.0)  |(-50.0,-50.0,152.0) |EA1   |
[2][1][2]|(-49.0,-150.0,52.0)   |(51.0,-50.0,152.0)  |EA1   |
[2][2][2]|(52.0,-150.0,52.0)    |(152.0,-50.0,152.0) |EA1   |

Objects Load (in milliseconds)
55541
Object  |Load|
albert  |5859|
luis    |6014|
someone |0   |

Second | Environment Agents          Second | Time of Service (in seconds)     Second | Average Region Load
55541        1                       55541        75                          55541        158

Second | Administered Objects        Second | * Time of Service used           Second | Time of Response
55541        3                       55541        14                          55541        1282

Object Positions
55541
Row/Column|X#-150_-50     |X#-49_51|X#52_152|
Z#-150_-50|               |        |        |
Z#-49_51  |               |        |        |
Z#52_152  |someone,albert|         |luis    |
```

Figure 6-37. The Environment Agent after 75 seconds since the beginning.

For this case study, none of the agents knows about the location of the others. The behavior of agents luis and albert was predefined and the avatar named someone does not move. The behavior of agent luis consists in walking forward until it collisions to either a wall or a box, then turns to the right 90 degrees, then walks forward one step and turns again to the right 90 degrees, then it continues walking forward; one time the avatar of luis turns to the right and the other to the left until a collision with the avatar of someone occurs; such event makes the behavior of luis to stop its execution. The behavior of agent albert is analogous to luis´ but albert first turns to the left instead of to the right.

In previous Figures 6-34 and 6-36 the avatars of luis and albert show graphical emotional state. When such agents start walking forward they set its state as "happy" and when a collision of an avatar with a box or a wall occurs, its agent changes its state to "fear" and so on. For this simulation, no emotional state affects agents´ behavior.

As long as the avatars move through the environment, the Environment Agent notifies to their corresponding agents, their current state and the state of objects around each one as illustrated in Figures 6-38 and 6-39.

```
luis
Agent ID: 3861745940699470963

L has received
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: update-environment-object
Decoding XML of EnvironmentObject: Avatar
position=(140.0,0.0,125.49771118164062)
Behavior continues
ACL invoking receive
Collision occured...
ACL has received
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: update-environment-object
Decoding XML of EnvironmentObject: Avatar
position=(140.0,0.0,125.49771118164062)
Behavior continues
ACL invoking receive
```

Figure 6-38. Agent luis knows its position x=140, y=0, z=125.

```
albert
Agent ID: 4611686017012208752

invoking receive
Collision occured...
ACL has received
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: inform
Behavior continues
ACL invoking receive
Waiting por position update
ACL has received
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: update-environment-object
Decoding XML of EnvironmentObject: Avatar
position=(-110.0,0.0,122.39806365966797)
Behavior continues
ACL invoking receive
```

Figure 6-39. Agent albert knows its position x=-110, y=0, z=122.

The simulation continues as illustrated in Figure 6-40 where agent albert reaches its goal as illustrated in Figure 6-41.

Figure 6-40. The scenario after 103 seconds since the beginning.

albert

Agent ID: 4611686017012208752

```
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: inform
Behavior continues
ACL invoking receive
Waiting por position update
ACL has received
Message sender: Environment Agent_=)_(10.0.5.206,4611686016681361
Processing performative: update-environment-object
Decoding XML of EnvironmentObject: Avatar
position=(-59.40077209472656,0.0,115.3981704711914)
Behavior continues
ACL invoking receive
Collision occured...
Object 0: albert
Object 1: someone
Change - > Collision flag: false
End of behavior
```

Figure 6-41. Agent albert reached its goal.

The simulation continues as illustrated in Figures 6-42 to 6-44.

Figure 6-42. The scenario after 165 seconds since the beginning.

Figure 6-43. The scenario after 188 seconds since the beginning.

Figure 6-44. The scenario after 220 seconds since the beginning.

Figure 6-45. The scenario after 250 seconds since the beginning.

The metrics of the Environment Agent at the same time of the Figure 6-45 are shown in Figure 6-46.

The situation reported by agent luis when it reaches its goal is shown in Figure 6-47.

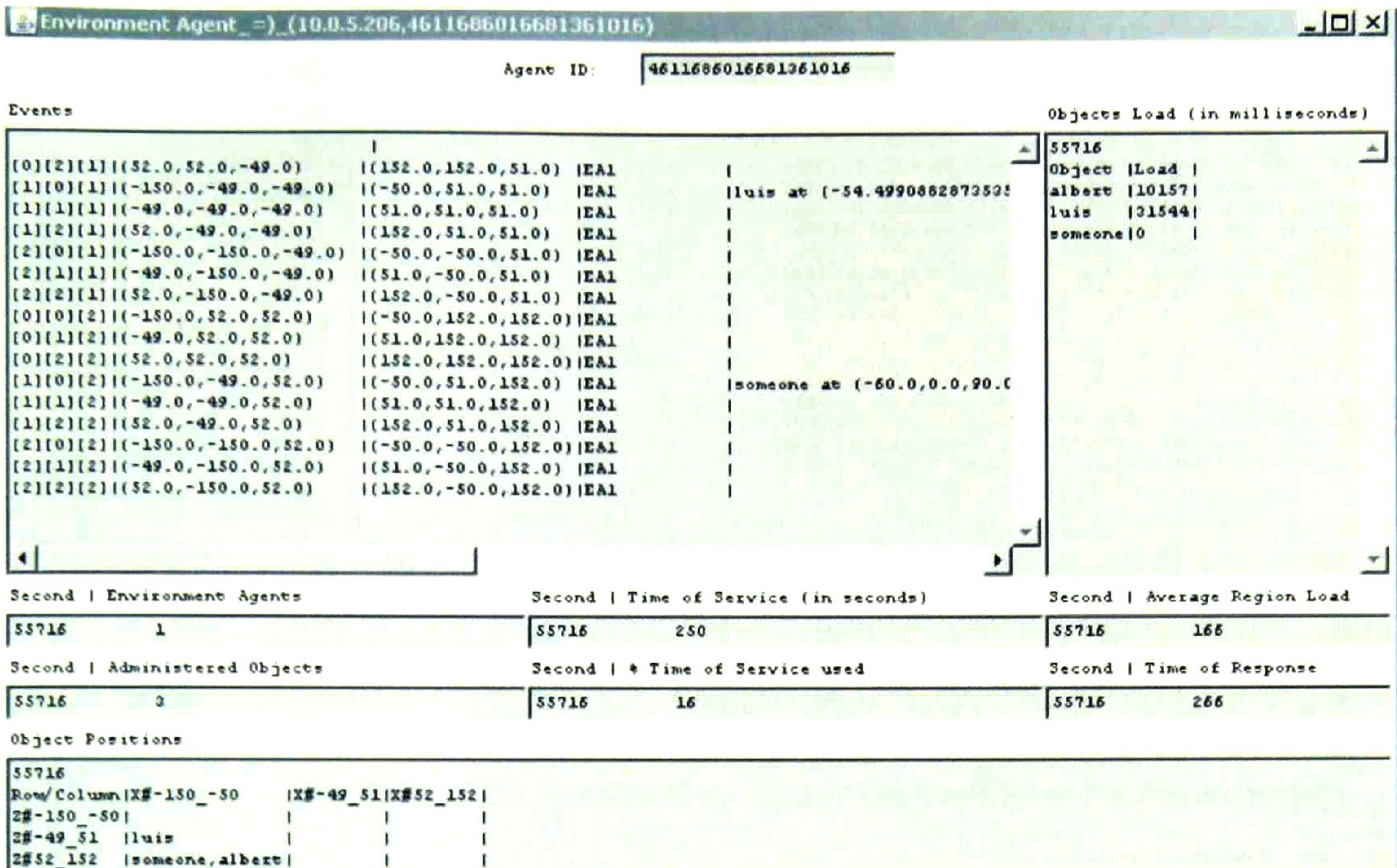


Figure 6-46. The Environment Agent after 250 seconds since the beginning.

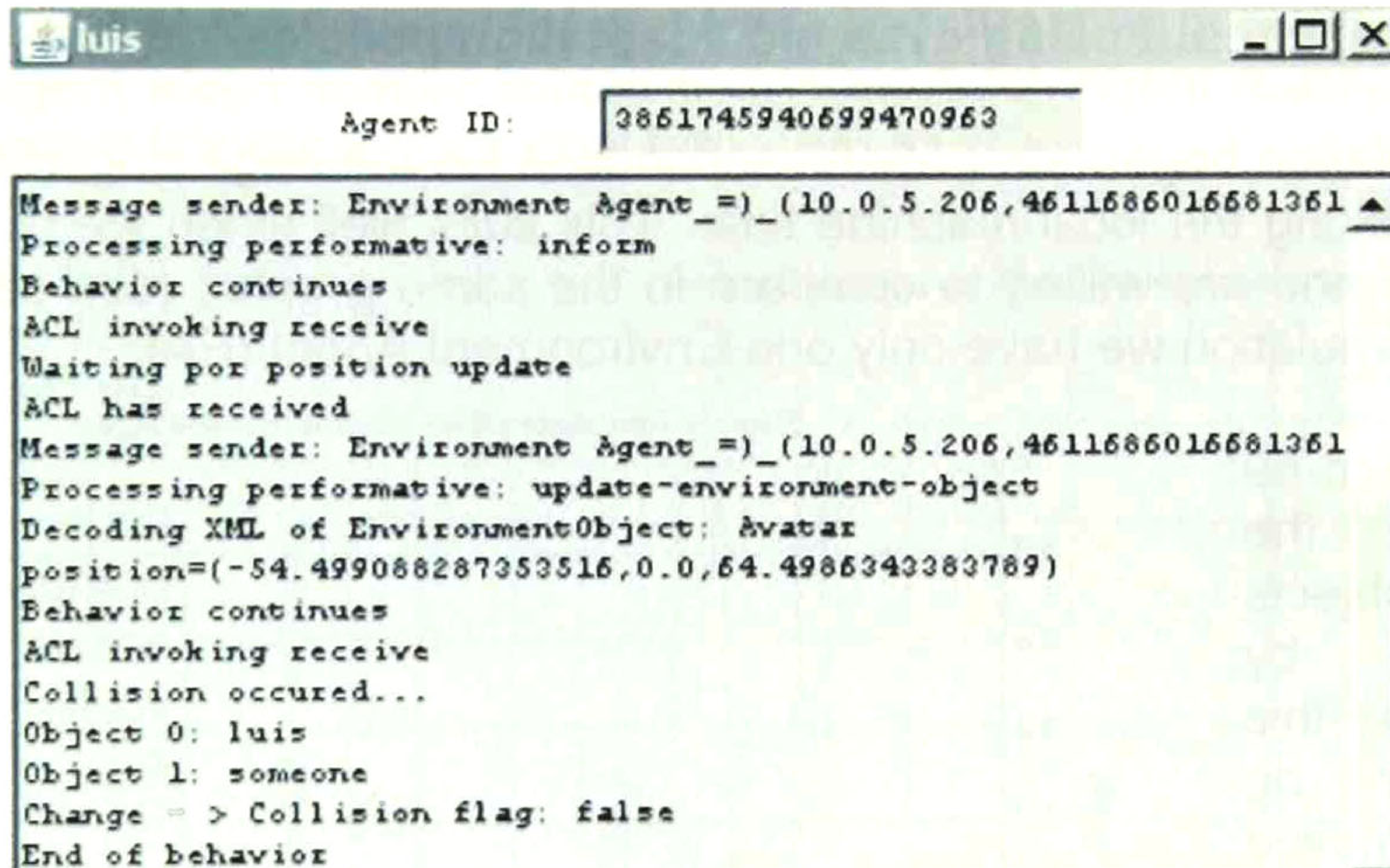


Figure 6-47. Agent luis reached its goal.

The metrics of the Environment Agent after no agent carries out any intention, that is, the end of the simulation is shown in Figures 6-48. Also the percentage of *time of service* of the EA begins to decrement. The latter situation would lead to a fusion if there were more than one Environment Agents representing the environment (see section 6.3.3).

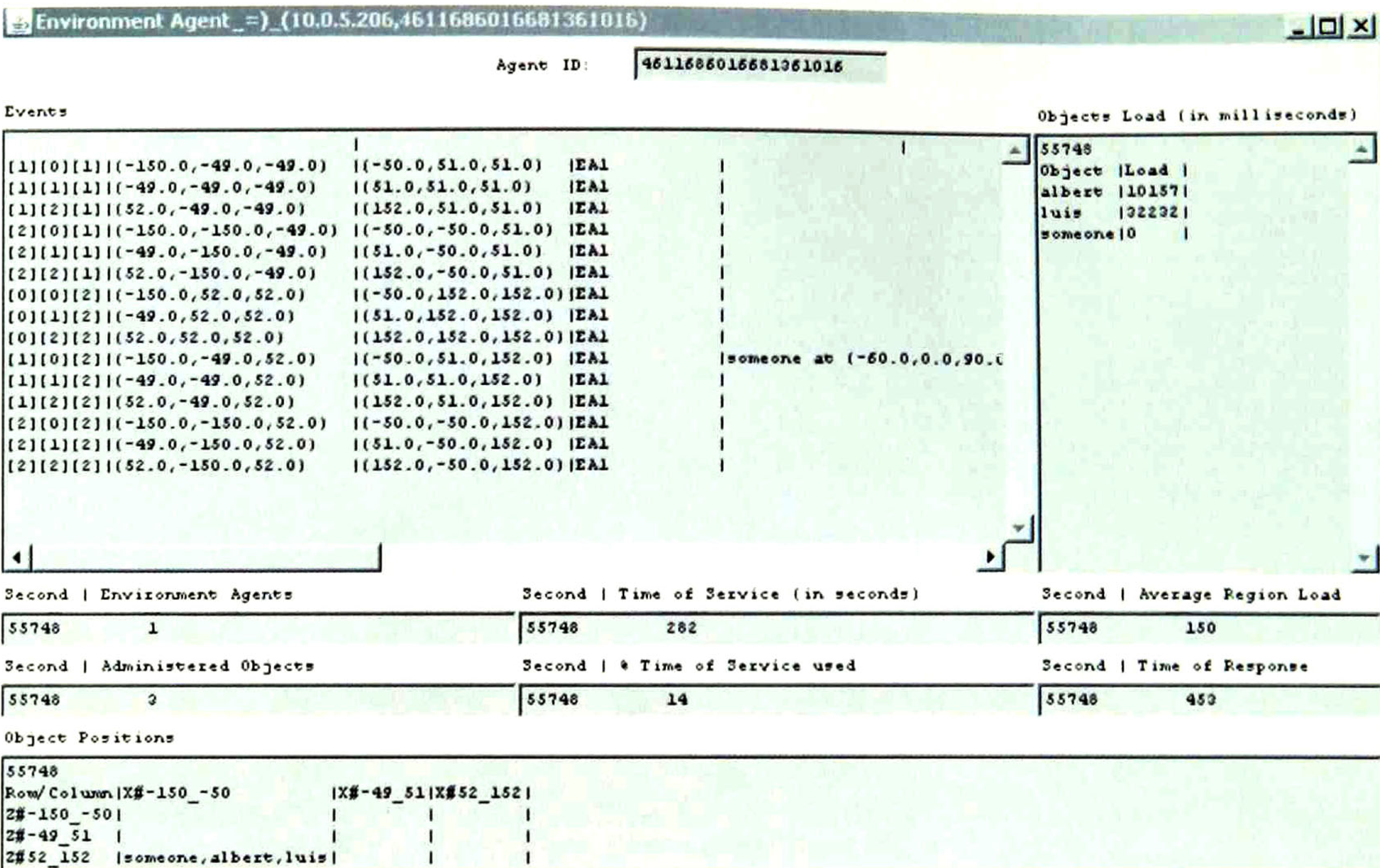


Figure 6-48.The Environment Agent after the end of the simulation of the second case study.

6.3.2 Metrics without using dynamic adaptation policies

This simulation began after 55485 seconds since the beginning of the day (see Figure 6-3) taking the local machine time. This suits well when we use more than one machine and are willing to compare in the same graphic (see section 6.3.3). During this simulation we have only one Environment Agent (EA).

In Figure 6-49 we depict the amount of objects administered by the EA. In this case study no object is removed from the environment, unlike the first case study (see Figure 6-20).

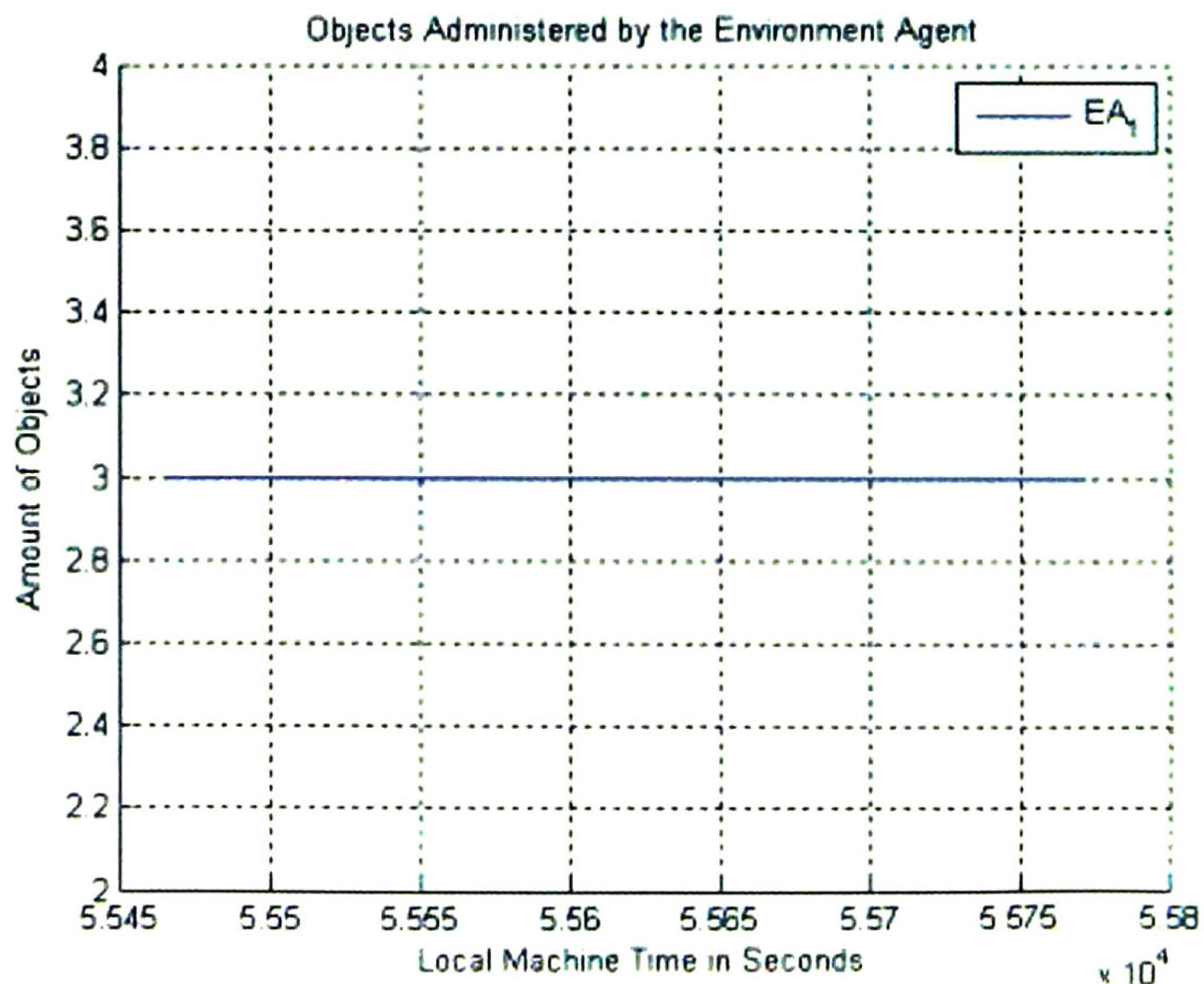


Figure 6-49. Objects Administered using one EA in the 2nd case study.

In Figure 6-50 we depict the average region load of the EA. The load of an object is the amount of *time of service* dedicated to process the intention of an agent that causes modifications to such object.

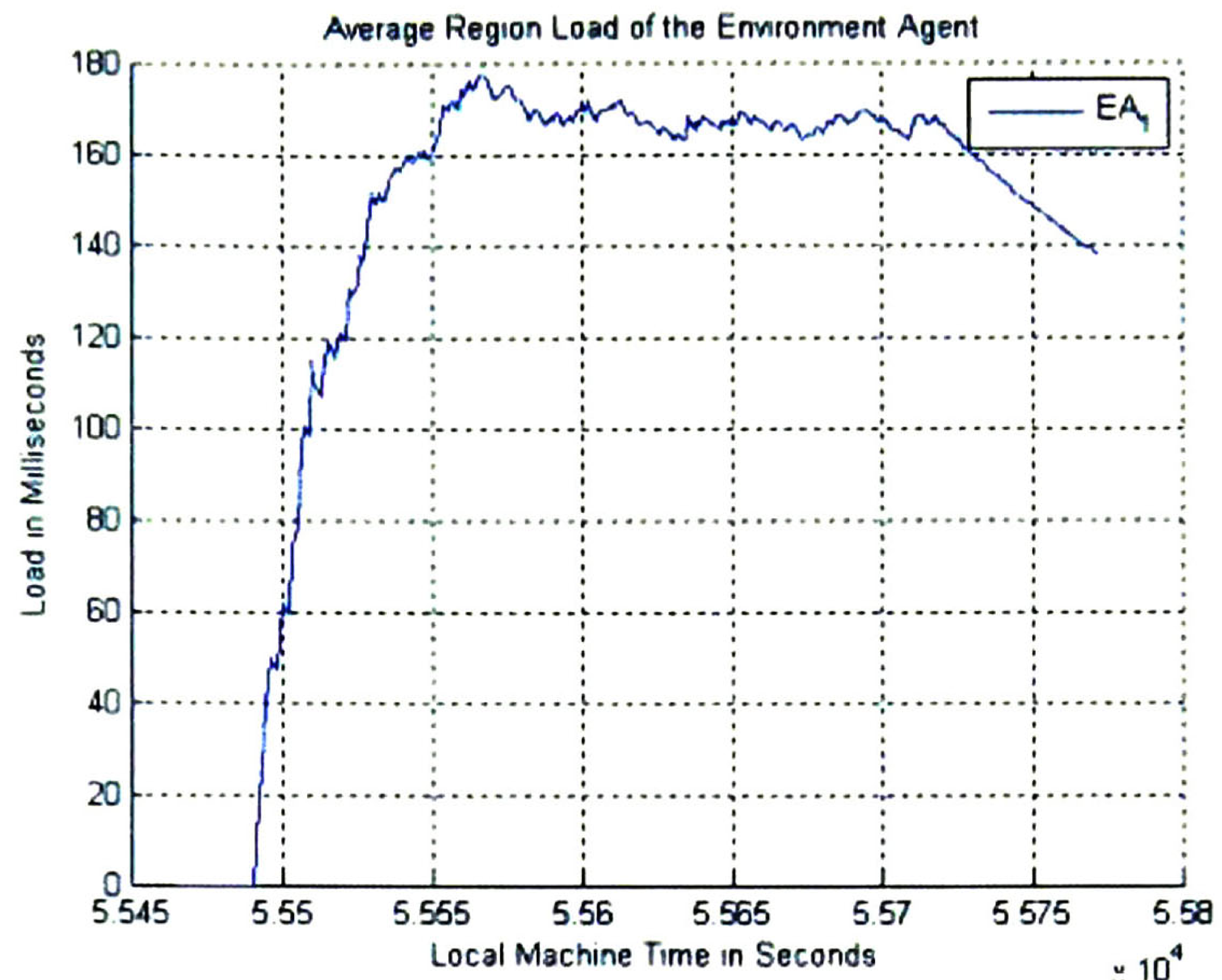


Figure 6-50. Average Region Load using one EA in the 2nd case study

The load of the *region* administered by the EA is the sum of the accumulated loads of all objects that are either members or neighbors of the *region* (see section 5.4). In this case study no object is removed from the environment, thus the load caused by agent albert is still counted even after such agent reached its goal. At time 55718 agent luis reached its goal too.

In Figure 6-51 we depict the *time of response* for the processed intentions, such time is counted since intention arrived to MailBox until the EA finished processing the intention. If such time increases considerably, this would cause the starting of a dichotomy process (see section 6.3.3).

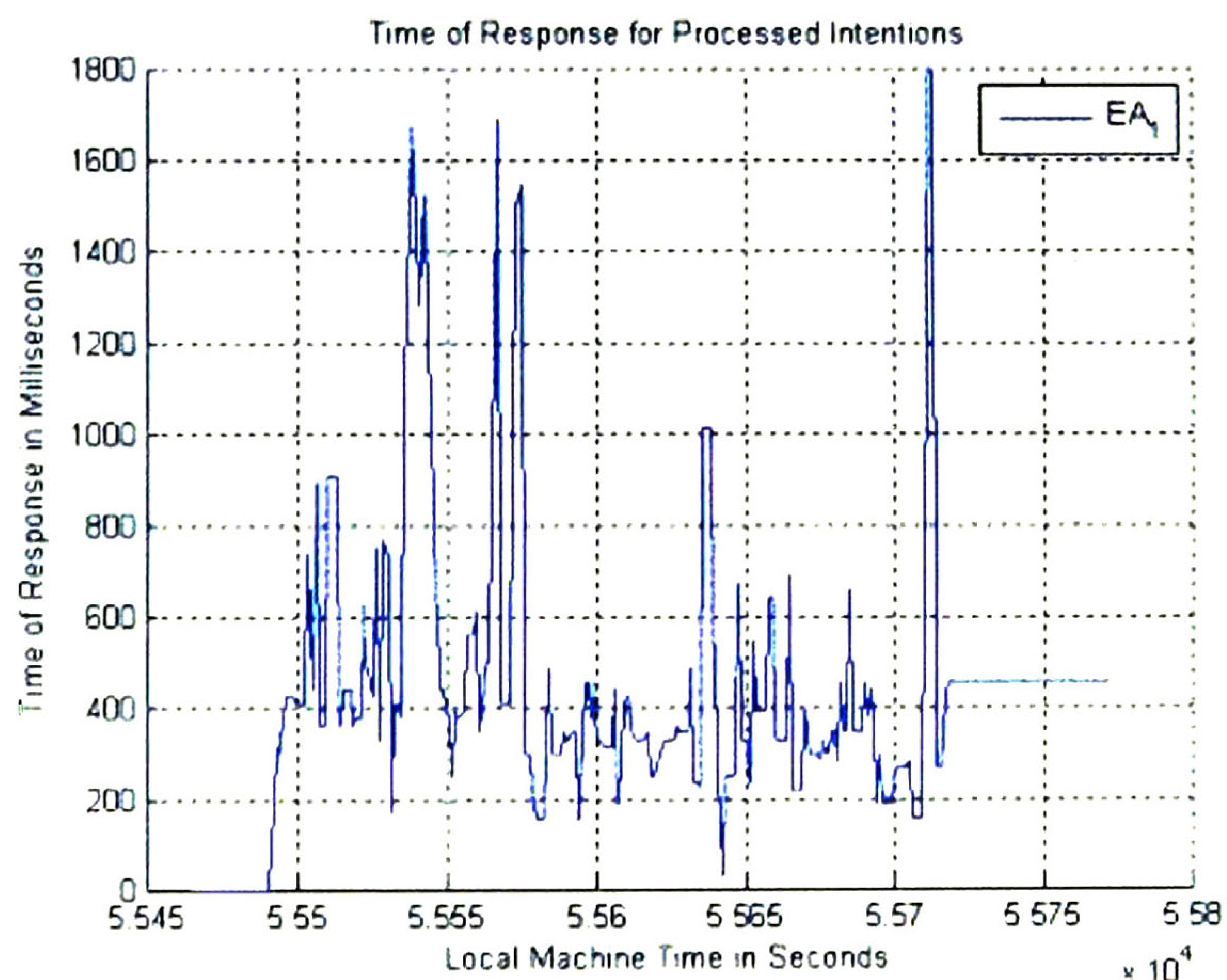


Figure 6-51. Time of Response using one EA in the 2nd case study.

In Figure 6-52 we depict the percentage of *time of service* that the EA dedicated to process agent intentions. After both agents luis and albert reached their goals, they did not send intentions to the Environment Agent anymore and, its percentage of *time of service* decreased.

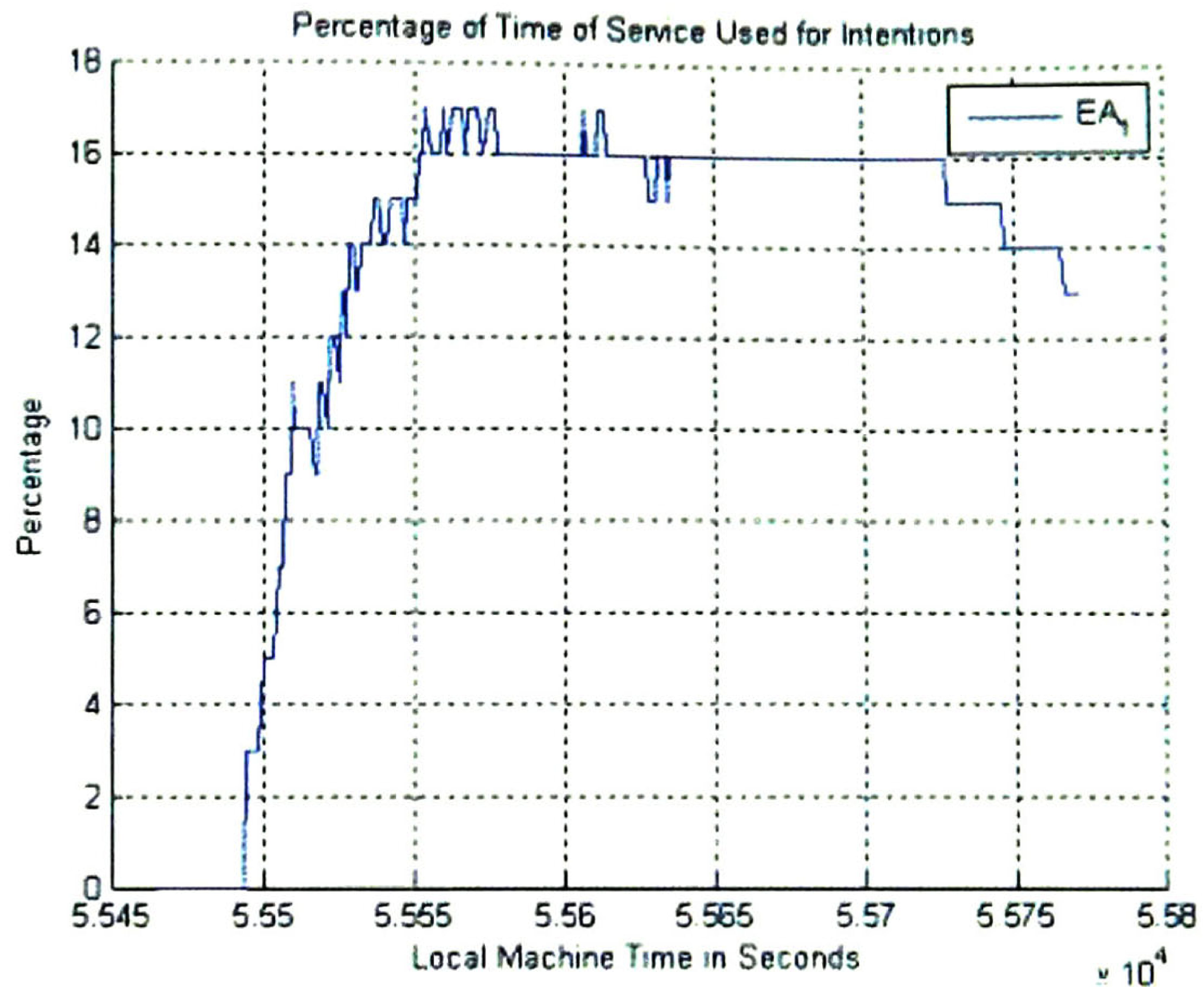


Figure 6-52 Percentage of Time of Service using one EA in the 2nd case study.

6.3.3 Metrics applying dynamic adaptation policies

In Figure 6-53 we depict the amount of agents running during this simulation. Two EAs are running after EA_1 has applied a dichotomy process. After approximately 60 seconds a fusion process is started and EA_1 sends its information to EA_2 which continues attending the entire environment.

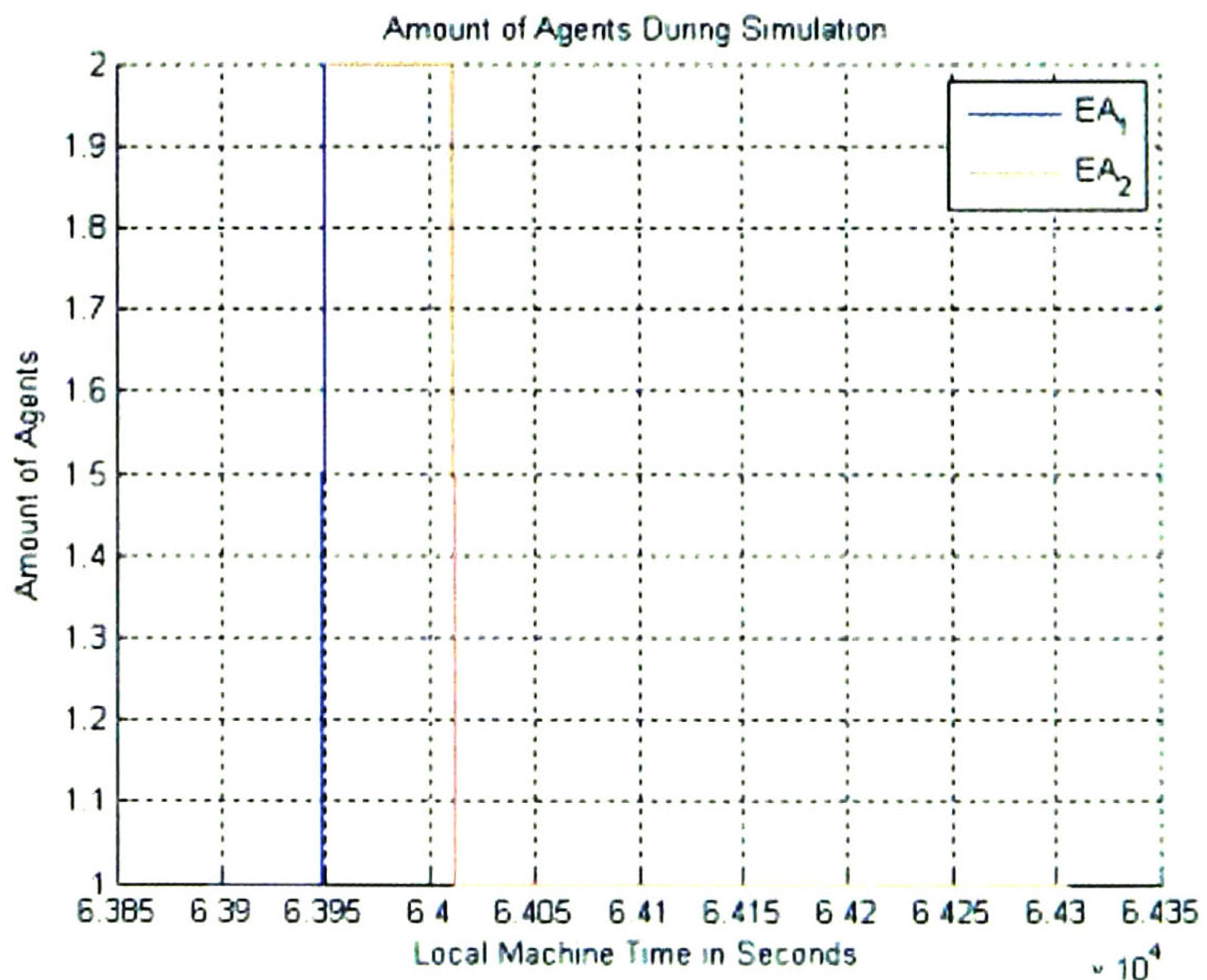


Figure 6-53. Amount of agents using up to two EAs in the 2nd case study.

In Figure 6-54 we depict the amount of objects administered by the EAs. At the beginning of the simulation the EA_1 administers 3 objects. After the dichotomy process, EA_1 administers 2 objects and EA_2 administers 1 object. At every moment the sum of all objects administered by the two EAs is 3.

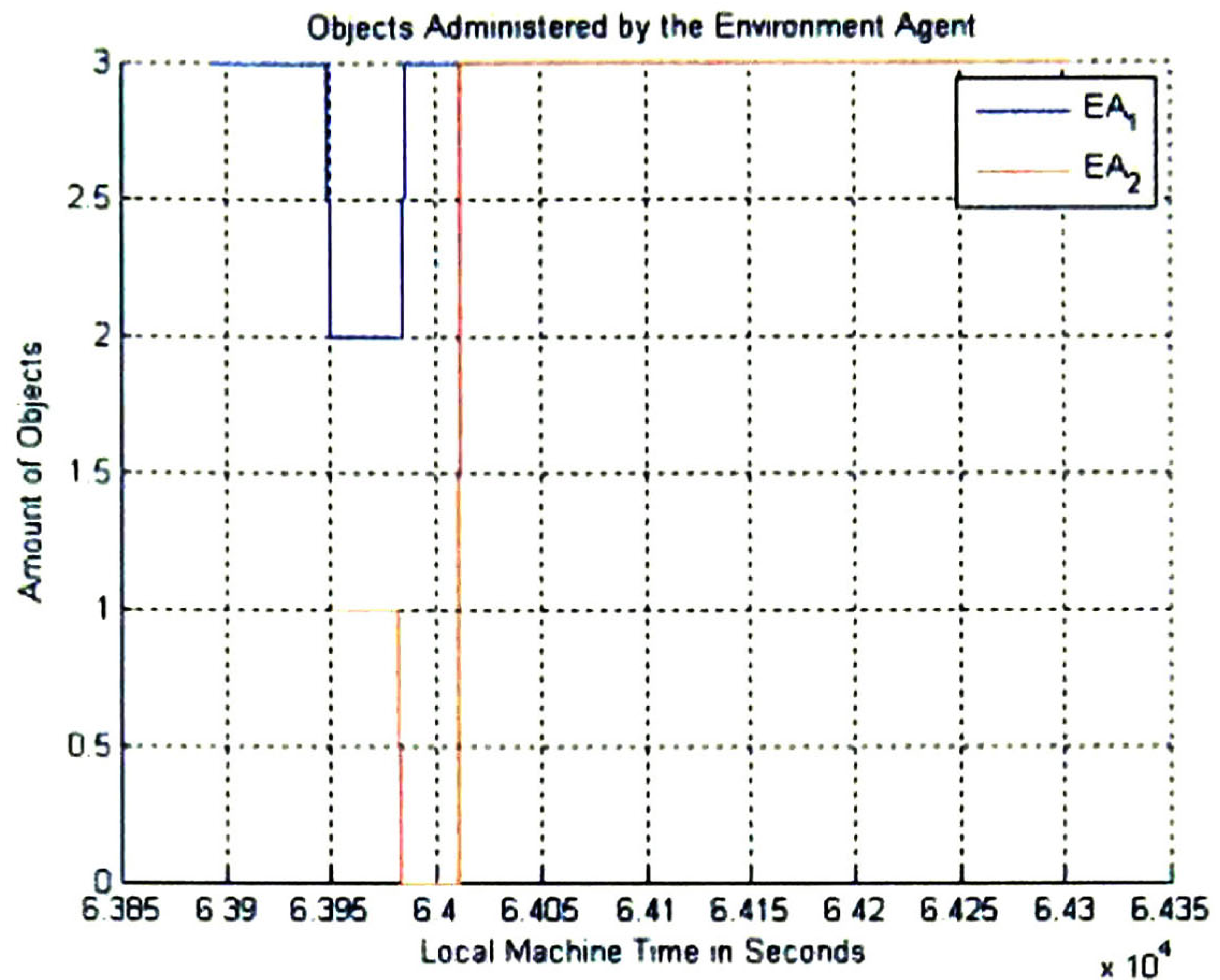


Figure 6-54. Objects Administered using up to two EAs in the 2nd case study.

In Figure 6-54, after some elapsed time, EA_2 reports that it administers no Environment Object and EA_1 administers 3 objects. This situation happened because at time 63983 the avatar of luis moved from the region administered by EA_2 to the region of EA_1. After time 63983, EA_1 stopped notifying EA_2 about luis' movements because no part of luis was occupying the region of EA_2. After time 64012 EA_1 carried out a fusion process with EA_2.

In Figure 6-55 we depict the average region load of the EAs. Comparing this Figure with 6-50 is notable that the load of around 180 milliseconds was shared among the two Environment Agents, being the maximum around 95 milliseconds.

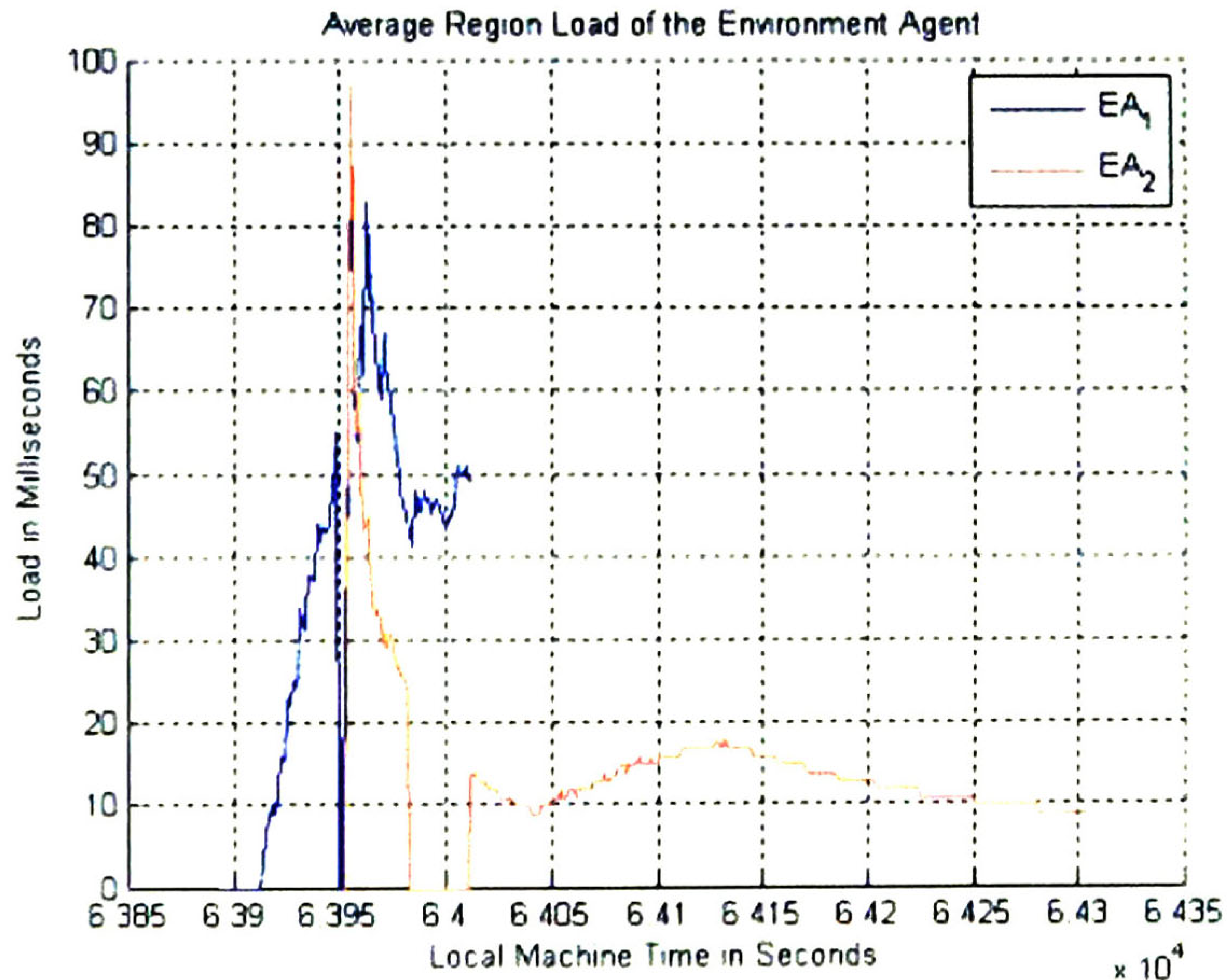


Figure 6-55. Average Region Load using up to two EAs in the 2nd case study.

In Figure 6-56 we depict the *time of response* for the processed intentions of the EAs. At time 63948 such time increased three consecutive times and, caused the starting of a dichotomy process (see also previous Figures 6-53, 6-54 and 6-55). This Figure shows a pending problem to be solved.

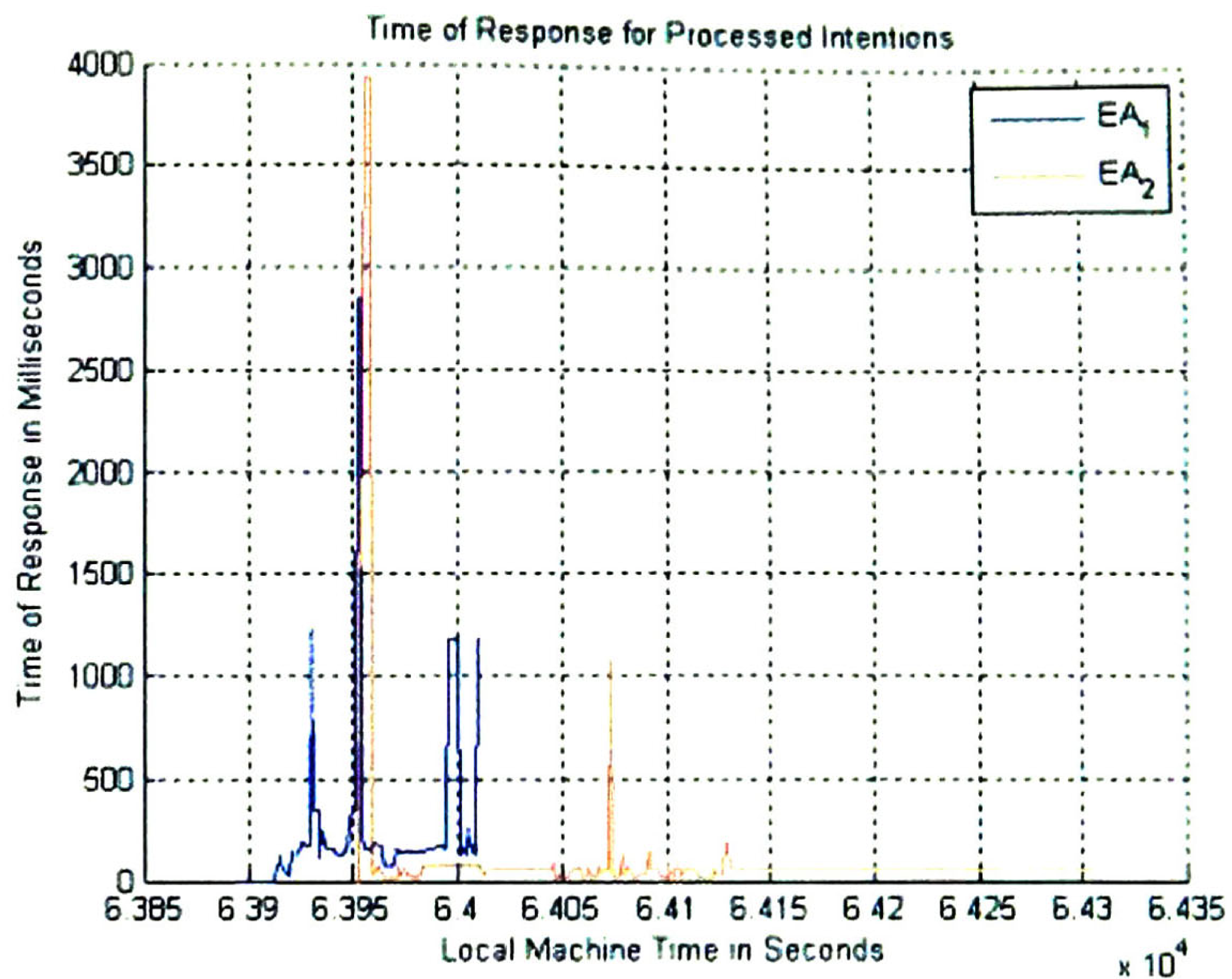


Figure 6-56. Time of Response using up to two EAs in the 2nd case study.

The previous Figure 6-56 shows slightly the pending problem to be solved described in section 6.2.3 about Figure 6-27.

In Figure 6-57 we depict the percentage of *time of service* of the EAs. In this case study, after the dichotomy process EA_1 neither EA_2 were close to 10% and, at time 64012 EA_1 decided to carry out a fusion with EA_2. After luis reached its goal, EA_2 did not received intentions anymore and, its percentage of *time of service* also decreased.

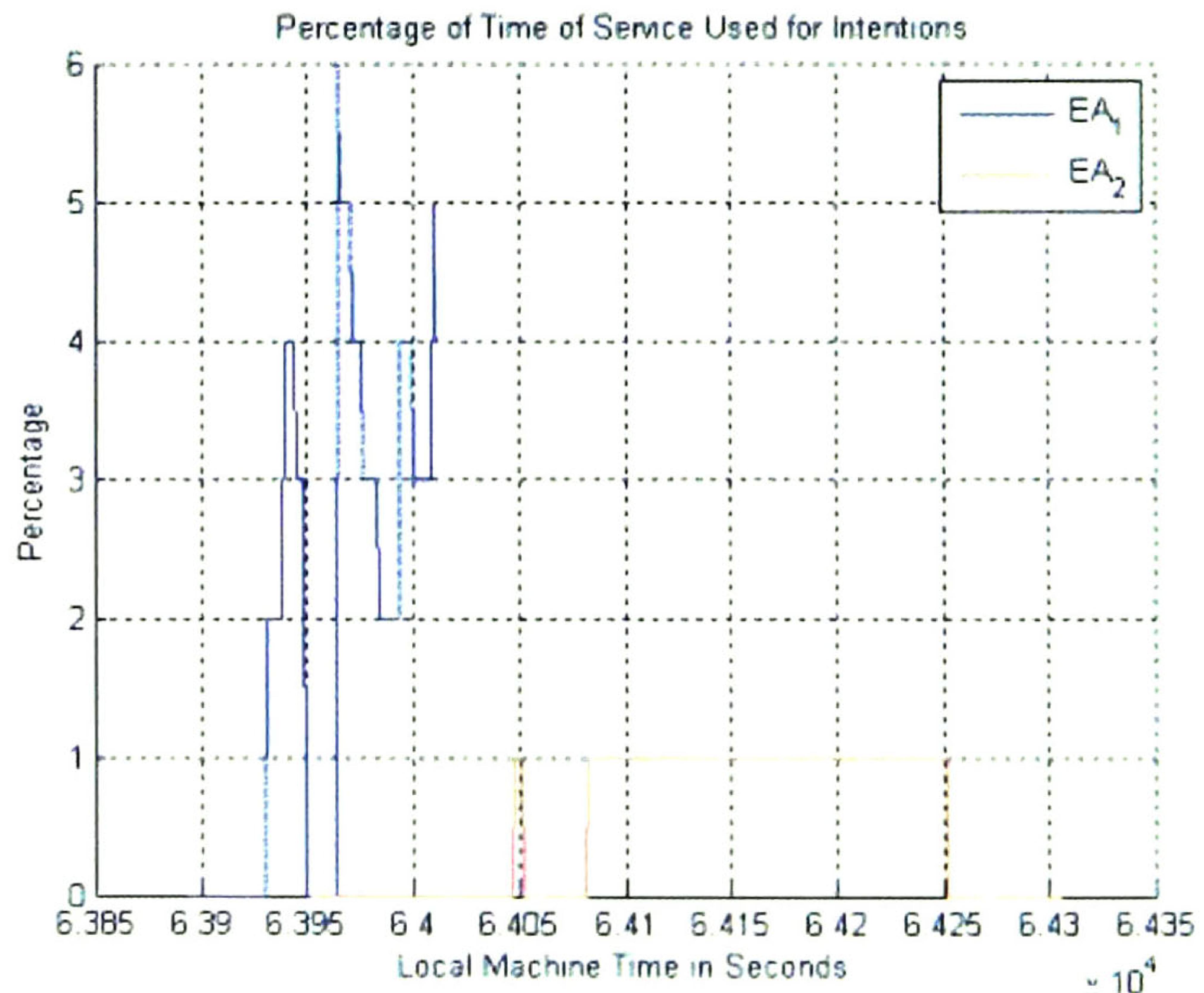


Figure 6-57 Percentage of Time of Service using up to two EAs in the 2nd case study.

Comparing the previous Figure 6-57 with 6-52, it is also notable that the percentage of *time of service* dedicated by these two Environment Agents was divided to less than half.

6.4 Prey Predator Avatar Chasing using the Scenario Descriptor

We present here the case study on the work developed about the Scenario Descriptor by [ZARAGOZA] with the Rendering of [MARTINEZ] for GeDA-3D. In this case there are more box type objects besides the boxes already included in the environment as described in previous section 6.3; see Figures 6-58 and 6-59.

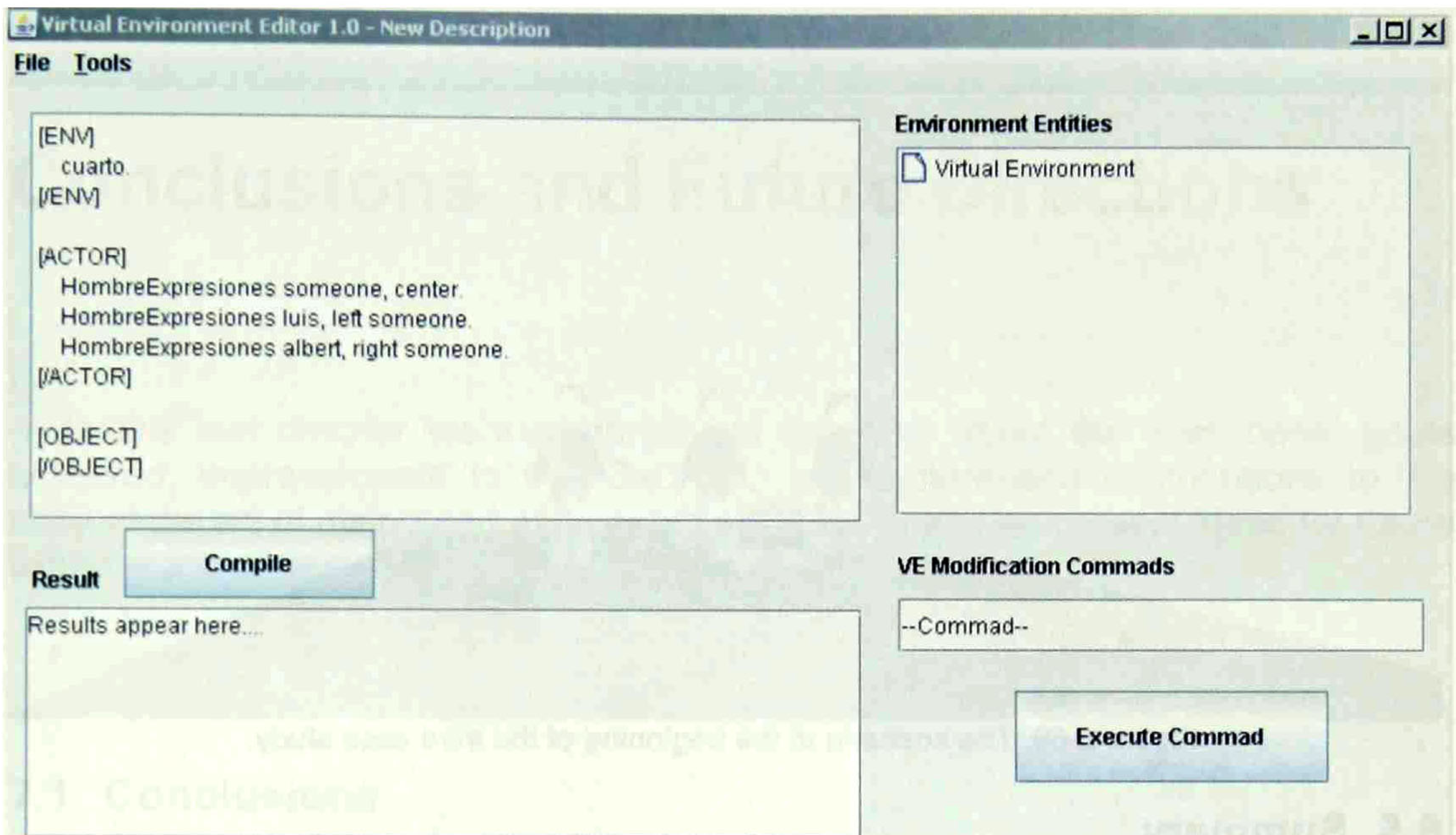


Figure 6-58. The Scenario Descriptor for the third case study.

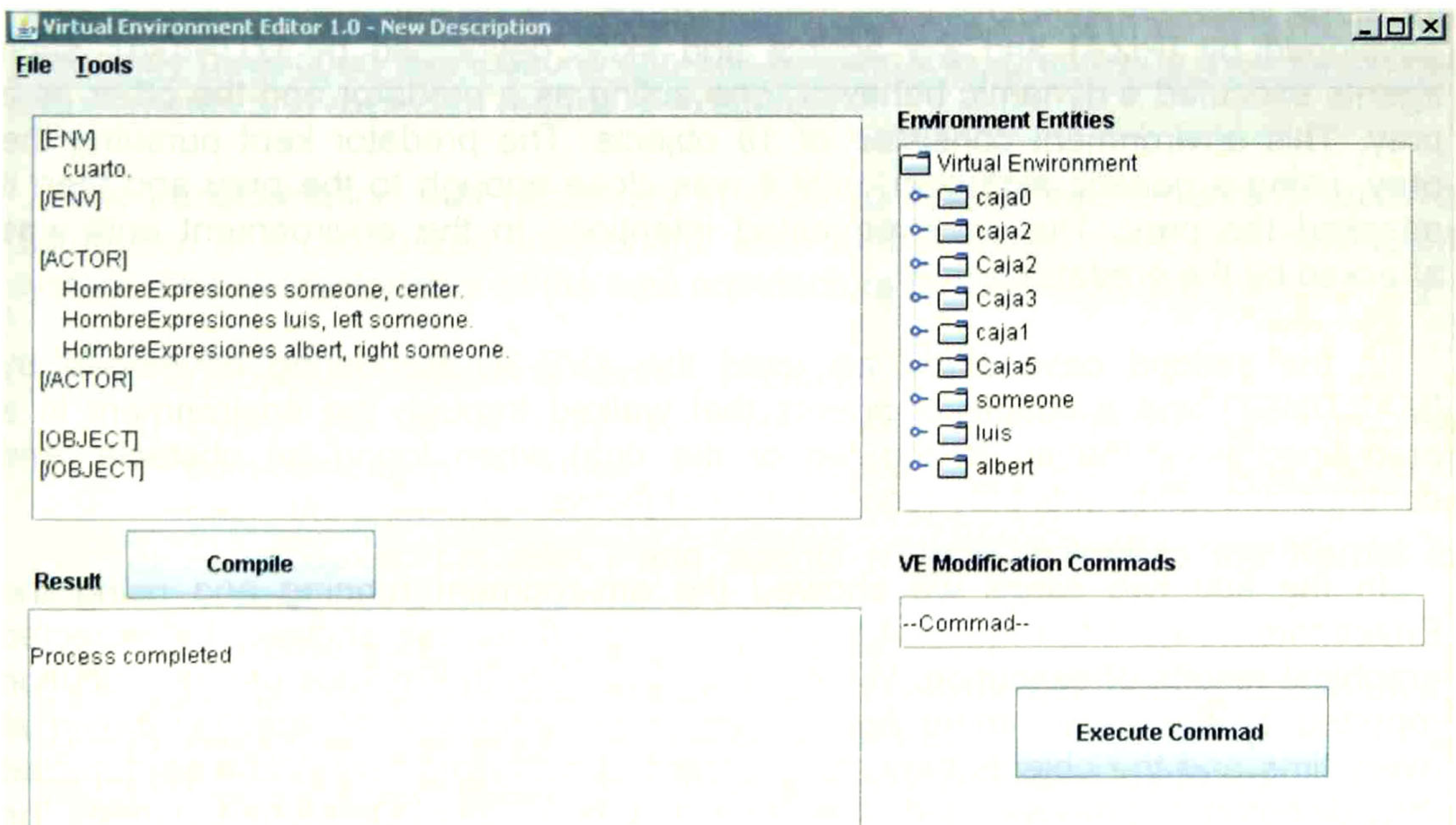


Figure 6-59. The Scenario Descriptor after compiling the described scenario.

At the moment of this writing this Scenario Descriptor only supports one type of avatar of the collection provided by AVE-3D. The evolution of the scene is analogous to the previous case study with a difference on the type of avatars used and the objects included as illustrated in Figure 6-60.

Figure 6-60. The scenario at the beginning of the third case study.

6.5 Summary

In the first case study we used the ViSCA Rendering and the Scene Descriptor developed by [PIZA] and the agents and skills developed by [ZUÑIGA]; such agents executed a dynamic behavior, one acting as a predator and the other as a prey. This environment consisted of 18 objects. The predator kept pursuing the pray, using a genetic algorithm, until it was close enough to the prey and then it attacked the prey. The prey requested intentions to the environment until was attacked by the predator.

In the second case study we used the AVE-3D Rendering developed by [MARTINEZ] and a couple of agents that walked through the environment in a predefined way, turning to the left or the right when found an obstacle, and continue executing until they reached their objective.

In the first two cases we showed the environment running and using the Environment Agent to request it to carry out intentions. We showed the expected graphical results of execution. We depicted in charts the metrics of the execution reported by the Environment Agents showing the amount of agents running at every time and the objects they administered at every moment. We also depicted the dichotomy process and the fusion process being started when the corresponding EA determined that it reached either its upper or lower bound limit.

Chapter 7

Conclusions and Future Directions

In this last chapter we summarize our thoughts about the work done, goals achieved, improvements to the GeDA-3D architecture and contributions to the state of the art of distributed multi-agent systems; finally we present ideas for future work.

7.1 Conclusions

The first goal of this work was, transforming the platform of GeDA-3D into a flexible and scalable platform to run different types of 3D applications, while the previously developed applications can still be runnable, preserving their functional requirements and expected results of execution.

The transformation was necessary in order to satisfy requirements to achieve distribution of a centralized component, such as the environment of a multiagent system. The characteristics of the new architecture for GeDA-3D are the following:

1. The proposed platform is flexible, because:
   a) Every Kernel is only in charge of: a mechanism for process communication, a limited administration of low level processes, and basic I/O channels;
   b) All modules, processes and agents may be linked to the Kernel in runtime.
2. The proposed platform is scalable, because:
   a) Every Kernel is in charge of only the processes and threads started on it;
   b) All kernels are aware of the others workload;
   c) Processes are loaded in the machine with the lowest workload.

d) Adding another machine will lead to load processes preferably in that machine due to that is the one with the lowest workload.

3. The proposed platform provides all general services of a middleware, presenting a set of operations for thread management, process addressing, peer to peer communication, group communication and message buffering easy to use for programmers.
4. The platform services are provided through the MicroKernel and its attached modules for Thread Management, Process Addressing, Message Storage-Delivering, Reliability, Distributed Mutual Exclusion and Group Addressing, which have high cohesion by themselves and are weakly coupled with the MicroKernel.
5. The Thread Management Module working together with the Process Addressing Module made easier the implementation of the Agent Management System that is in charge of providing a white page service. That is because it was only needed the implementation of the Agent Management Module, in charge of managing a few agents´ issues and, using the operations provided by the first two modules to provide the complete service.
6. The use of the ASN and the AUN for agents, managed both by the Agent Management System made easier the implementation of the Directory Facilitator, because every agent is registered as a server using an ASN and optionally may request to register as an individual using its AUN.
7. The Message Storage-Delivering, Reliability, Distributed Mutual Exclusion and Group Addressing Modules made easier the implementation of the ACL Message Service. That is because the latter is only in charge of providing the standardization of ACLMessages content, treat messages analogously to the OSI transport layer and, providing each agent with a Listener thread to receive ACLMessages.
8. After the MicroKernel substituted the old Message Transport Service of the previous version, the previously developed applications can still be runnable including their expected results.

The distribution of a centralized component is achieved through providing the following facilities:

1. Addressing of every process in the system using its Process Service Name (PSN) by default, that is, if there are various processes that can provide the same service, any of them is eligible to execute a required task.
2. The creation of a Process Unique Name (PUN) permits processes to communicate just with the process that owns that PUN if the latter is the best eligible option to provide a required service. The latter two are provided by the Addressing Module.
3. Group management lets various processes to join a group in the system and then, receive messages from the others in order to synchronize their activities. This is provided by the Group Addressing Module.
4. Given that all instances of Kernel in different machines notify the others its workload, if a specific process requires distributing its workload with a new

process, then when requesting to load a new process, it is chosen the machine with the lowest workload and, the performance of the service is improved. This is provided by the MicroKernel through the PlatformFinder.

5. A mechanism to store messages until they are successfully delivered permits to obtain a message and if necessary resend such message to another destination; the latter is provided by the Storage Module.
6. Reliable communication permits doing the best effort to localize a destination and deliver a message as long as an appropriate destination is traceable; the latter is provided through the Reliability Module.
7. Given that a PUN is an extension of a PSN, if a process *P* is member of a group of processes that use the same PSN and, a process *Q* sends messages to *P* using a PUN, when *P* is no longer providing service the Kernel retries delivering the message using the corresponding PSN.

The final goal of distributing adaptively multiagent environments was met as follows:

1. We use special agents named Environment Agents as a set of one or more agents that represent the entire environment as a unit.
2. An Environment Agent (EA) only represents one part of the environment.
3. The Scenario Module (MS) manages a collection of Environment Objects and, it is eventually partitioned producing an Environment Partition that consists in a subset of such objects.
4. The Context Module is expected to be used (see section 7.2) as the source for a cached copy called ContextCache which contains part of the information of the Context Module.
5. The modules Scenario (MS) and Context (MC) were distributed through available processors (or machines) over the network being allocated their corresponding partitions and caches in different Environment Agents.
6. Every EA owns an Environment Partition and a ContextCache and a DistributionMap.
7. The DistributionMap is used as a shared memory for all the Environment Agents.
8. Every EA uses its DistributionMap to know which Environment Agent is in charge of which region of the environment in order to communicate with the appropriate agent when necessary.
9. The EA also uses the DistributionMap to know if it is required to send to other Environment Agents a message regarding an Environment Object which is on the border of two regions.
10. The distribution of the modules MS and MC through available processors is achieved through exploiting the platform services for distribution of a centralized component described above.

Finally, the compliance of the requirements for the middleware is detailed in the following table:

#	Requirement	Satisfied through	Details in
1	Completely distributed execution	Middleware	Chapter 4
2	Open source	Source code	Chapters 3, 4, 5 and [RAMOS, PIZA, ZUÑIGA, AGUIRRE, MARTINEZ, and ZARAGOZA]
3	FIPA compliant architecture	Middleware	Chapter 4
4	Easy to use and extend for programmers and non programmer users	Scenario Descriptor and Scene Descriptor	Chapter 3 and [ZARAGOZA]
5	Agent replication and mobility capabilities	Pending	Chapter 3
6	Complete and dynamic agent administration (creation, starting, stopping, addition at runtime)	Agent Administrator	Chapter 3 and [ZUÑIGA]
7	Agent goals setup	Scene Descriptor	Chapter 3 and [PIZA]
8	Agent skills assignment	Scene Descriptor and Agent Administrator	Chapter 3, [PIZA and ZUÑIGA]
9	Private agent knowledge	Agent Architecture	Chapter 3 and [ZUÑIGA]
10	Human-agent interaction	Agent Administrator	Chapter 3 and [ZUÑIGA]
11	Assignment of personality and emotional state to agents	Scene Descriptor Agent Architecture	Chapter 3, [PIZA, and ZUÑIGA]
12	Environment simulation	Environment Agents	Chapter 5
13	Multiple environment executing concurrently	Middleware	Chapter 4
14	Agent assignment to each avatar (a graphical virtual entity)	Scene Descriptor	Chapter 3 and [PIZA]
15	Avatars database administration	Agent Administrator	Chapter 3 and [ZUÑIGA]
16	Environment-Render direct interaction	Environment Agents and Rendering	Chapter 5 and [MARTINEZ]
17	Virtual environment collision event handling	Rendering	[PIZA] and [MARTINEZ]
18	Unified shape description of agents between render and agent private knowledge	Agent Architecture	Chapter 3, [PIZA and ZUÑIGA]
19	Sensors and effectors simulation	Agent Architecture	Chapter 3

#	Requirement	Satisfied through	Details in
20	Transparent communication between all modules and all entities	Middleware	Chapter 4
21	Scene evolution control, validating each agent intention to change virtual world	Environment Agents	Chapter 5

The proposed reengineering of GeDA-3D architecture was the key to provide the basis for building the distributed environment. Once GeDA-3D platform was updated, it was easier the implementation of the distributed environment, because the environment agents now have to deal only with policies related to the environment representation and scene evolution. All issues concerning to processes, groups, multi-environment support and communication were delegated to the distributed system platform.

7.2 Future Directions

As future work we propose the following ideas:

1. The platform could implement the mobility service, through weak mobility using java serialization, to move a process from one machine to another, loading only the state of variables in the process and, restarting the process on the target machine in one of a predefined set of states.
2. The platform could use network multicast for group messages, in order to interrupt only to the appropriate machines when sending messages.
3. The platform could provide guarantee delivering messages in order of time to all group members.
4. The platform could provide a more precise calculation of workload of machines. For instance considering: processes other than the JVM running the GeDA-3D platform, implementation details of the JVM about thread priority, etc.
5. The platform could provide dedicated communication channels, for critical services that should respond with the highest speed. An example is the case when the Group Addressing Module requires that the Critical Coordinator process grants the permission to create a group. Every process in the platform receives a message after Kernel has delivered messages that arrived before for other processes. The Kernel sends the message of a process after it has sent all messages from other processes that requested to do so before. So, to receive and send a message, the Critical Coordinator has to wait until the other processes have been attended and, group management is frequently used by the Environment Agents during scene evolution. The same situation occurs for messages sent among Environment Agents for coordination issues during fusion and dichotomy

processes. Also the Environment Agents working together with distributed Rendering modules might require such dedicated communication channels.

6. The platform could provide a mechanism in the Addressing Module in charge of providing the forwarding of messages delivered to a process *P* to another process *Q* that uses the same PSN and, causing that future messages sent by the process *S* were sent preferably to process *Q* instead of process *P*.
7. Analogously to Web Services, the platform could provide a service where processes would register some of their operations. Such operations would be used hiding details of knowledge of PSNs. An extension would apply for the yellow page service of the Directory Facilitator in the case of services provided by agents.
8. The platform could provide a better human agent interaction interface analogous to the script in the Scene Descriptor in order to communicate with the agent and suggest it actions to do.
9. The Context Module could be implemented as an independent entity apart from the Virtual Environment Editor. This would involve the analysis of both the Virtual Environment Editor to decouple the instructions in charge of validating rules on objects, including such instructions in the Context Module and, extend such rules to include the effect of object interactions.
10. The ContextCache could really serve as a cached copy of information stored in the Context Module and, communicate to it in order to bring from it all information related to the Environment Objects administered by the Environment Agent.
11. Every Environment Agent could have a redundant running copy of itself for robustness purposes and such copy should receive all intentions that the original received.
12. The Environment Agent could calculate the appropriate size of the Environment Cubes to be administered according to the size of the Scenario and the initial arrangement of the Environment Objects in the Scenario.
13. The Environment Agent could administer a dynamic size of Environment Cubes instead of the initial fixed sized cubes, that is, resizing a subset of cubes allocated in the same pair of coordinates, for example, for all the Environment Cubes which center was at the same *x* and *y* coordinate resize their width in the *z* coordinate.
14. The Environment Agent could request platform to send a terminate signal to an agent that controls an avatar which has been removed from the environment. For instance in the case study with two ships, the terminate signal is necessary because an agent could behave as an orphan, that is, the agent could waste cpu time and cause the sending of intentions to the Environment Agent about an avatar that no longer exists.
15. The criteria used by the Environment Agent for the dichotomy process could be refined. For that, the EA could also consider the load of its machine, the average load of the EA, the amount of administered objects, the relationships between agents and objects, etc.

16. The Environment Agent could consider agent interactions with a subset of objects, in order to group them in the same region when it is required to carry out a dichotomy process.
17. Once sensors and effectors were provided to ordinary agents, the Environment Agent could use the agent sensors and effectors to determine the proximity among such agent and other agents or objects around it instead of the calculation of proximity using the center of objects currently used.
18. Only when an entity (an object or agent) is near to an agent, the Environment Agent sends update messages to such agent about that entity. When an agent moves away from such entity and the agent cannot sense the entity, the entity's state and position remain in the agent's memory. The Environment Agent could be aware of each agent memory in order to refresh the memory of the agent when such agent comes back to the same place of the environment and such entity was moved from its original position. For example, an agent could sense the presence of cookies in a table in the kitchen in a time t_1 but, at this moment is not interested in eating cookies and decides to leave the kitchen, in a time t_2 the agent gets hungry and decides to come back to the kitchen expecting to find cookies according to its memory, which has not been refreshed because the agent is far from the cookies and, in a time t_3 when the agent is near enough to sense the cookies, the Environment Agent could refresh the agents memory informing that such cookies no longer exist, at least in the position the agent expected.
19. When the amount of created groups for every object sensed by agents was too high, the Environment Agent could administer a common memory for various agents, arranging the amount of groups to handle for notifying updates to the environment.
20. In a determined environment design, some agents could send to the Environment Agent a lot of intentions too simple that just disturb the management of computer resources from the Environment Agent. The Environment Agent is designed to be in charge of validating rapidly even complex intentions that would take enough computer resources. The platform implementation could be improved considering this situation. The latter would be important in order to avoid all Kernels sending excessive messages to the network that cause a fine-grained parallelism [TANENBAUM]. Such improvement would help to avoid network dependency, because now an Ethernet could turn into a bottleneck.
21. The platform could be used for simulation of real life problems like: urban traffic control, fire fighting, etc.

Bibliography

[AGUIRRE] Alonso Aguirre G. “Núcleo de GeDA-3D” Master thesis, February 2007.

[BADRINATH] B. R. Badrinath and T. Imielinski. “Replication and Mobility” Department of Computer Science, Rutgers University.

[CHIKOFSKY] Chikofsky, E.J.; J.H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy" IEEE Software January 1990 pp.14–15.

[COULOURIS] George Coulouris. “Distributed Systems, Concepts and Design” University of London and Cambrigde University, 2001.

[EAGER] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “Adaptive load sharing in homogeneous distributed system” IEEE Transactions on Software Engineering, vol. 12 no. 5, pp. 662-675, May 1986.

[ELMERHEBI] Souad Elmerhebi, Patrice Torguet, Nancy Rodriguez, Jean-Pierre Jessel. “The Effect Management in Distributed Virtual Environments” IRIT - Institut de Recherche en Informatique de Toulouse, France.

[EMMERICH] Wolfgang Emmerich, “Software Engineering and Middleware: A Roadmap” International Conference on Software Engineering, Limerick, Ireland, Pages: 117 129 ISBN:1-58113-253-0 2000.

[FERRARI] D. Ferrari and S. Zhou. “A load index for dynamic load balancing” Proc. Fall Joint Computer Conf., Dallas, TX, ACM-IEEE, Nov. 1986, pp. 684-690.

[FIPA] The Foundation for Intelligent Physical Agents. http://www.fipa.org/

[FIPA_ACL] FIPA ACL Message Structure Specification http://www.fipa.org/specs/fipa00061/SC00061G.html

[FIPA_SPEC] FIPA Agent Management Specification. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/specs/fipa00023/

[FUNKHOUSER] Thomas Funkhouser. "RING: A Client-Server System for Multi-User Virtual Environments" AT&T Laboratories, NJ.

[KREMIEN] Orly Kremien, Jeff Kramer. "Methodical Analysis of Adaptive Load Sharing Algorithms" IEEE Transactions on Parallel and Distributed Systems, Vol 3, No. 6, November 1992.

[KRUEGER] P Krueger and M. Livny. "The diverse objectives of distributed scheduling policies" in Proc. IEEE Int. Conf DCS, IEEE, 1987, pp. 242-249.

[LARMAN] Craig Larman. "UML and Patterns" Prentice Hall Pearson.

[LIVNY] M. Livny and M. Melman. "Load balancing in homogeneous broadcast distributed systems" Proc. Conf. Performance, ACM, 1982, pp. 47-55.

[MARTINEZ] Alma Verónica Martínez González. "Lenguaje para Animación de Criaturas Virtuales" Master thesis, August 2005.

[MCLLROY] M. D. McIlroy. "Mass Produced Software Components" In P Naur & B. Randell, editors, Software Engineering, Report on a conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, pp. 138-150., volume 1, pages 138–150.NATO Science Committee, Garmisch, Germany, 1969.

[ODELL] Odell, J., Parunak, H.V.D., Fleischer, M., Breuckner, S.: "Modeling Agents and their Environment" Agent-Oriented Software Engineering III, Giunchiglia, F., Odell, J., Weiss, G. (eds.) Lecture Notes in Computer Science, Vol. 2585. Springer-Verlag, Berlin Heidelberg New York (2002).

[ORACLE] http://www.oracle.com/technetwork/java/mvc-140477.html.

[PACHECO] Miguel Carlos Pacheco Alfonso Goulão; "Component-Based Software Engineering: a Quantitative Approach" PhD thesis, 2008.

[PIZA] H. Ivan Piza D. "Describing Interactions in Virtual Scenes Using a Declarative Description" PhD thesis, July 2007.

[RAMOS] Félix F Ramos, Fabiel Zúñiga, H. Iván Piza. A 3D–Space Platform for Distributed Applications Management. In proceedings of ISSADS 2002. (International Symposium of Advanced Distributed Systems). Guadalajara, Jal., Mex. November 2002. pp.66-77 ISBN 970-27-0358-1.

[SCHMIDT] Douglas C. Schmidt. "Why Software Reuse has Failed and How to Make It Work for You" C++ Report, vol. 11, no. 1, 1999.

[SMITH77] Smith R.G. "The CONTRACT NET: a formalism for the control of distributed problem solving" Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77),Cambridge, MA, 1977.

[SMITH80] Smith R.G. "A framework for Distributed Problem Solving" UMI Research Press 1980.

[SZYPERSKI] Clemens Szyperski. "Component Software Beyond Object-Oriented Programming" 2nd ed. Addison-Wesley / ACM Press.

[TANENBAUM] Andrew S. Tanenbaum. "Distributed Operating Systems" 1st ed.. Vrije Universiteit Amsterdam The Netherlands, 1995.

[TANENBAUM_STEEN] Andrew S. Tanenbaum, Maarten Van Steen. "Distributed Systems, Principles and Paradigms" 2nd ed. Upper Saddle River, New Jersey, Prentice Hall, 2007

[THEIMER] M. M. Theimer and K. A. Lantz. "Finding idle machines in a workstation-based distributed system" IEEE Trans. Software Engineering, vol. 15, no. 11, pp. 1444-1458, Nov. 1989.

[VOSINAKIS] S. Vosinakis, G. Anastassakis, T. Panayiotopoulos. "DIVA: Distributed Intelligent Virtual Agents" Knowledge Engineering Laboratory, Department of Informatics, University of Piraeus, Greece.

[WEYNS] Danny Weyns1, H. Van Dyke Parunak2, Fabien Michel3, Tom Holvoet1, and Jacques Ferber3. "Environments for Multiagent Systems State-of-the-Art and Research Challenges" 1) AgentWise, DistriNet, K.U.Leuven, B-3001 Leuven, Belgium; 2) Altarum Institute, Ann Arbor, MI 48105-1579, USA; 3) LIRMM, CNRS, Montpellier, 34392 Montpellier Cedex 5, France.

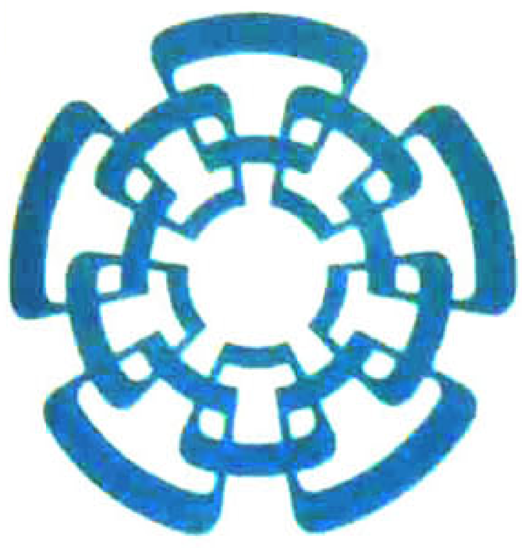
[WOOLDRIDGE] Michael Wooldridge. "An Introduction to Multiagent Systems" Department of Computer Science, University of Liverpool, UK. John Wiley & Sons, LTD 2004.

[ZARAGOZA] Jaime A. Zaragoza R. "Declarative Modeling Based on Knowledge" PhD thesis, December 2009.

[ZHOU87] S. Zhou. "An experimental assessment of resource queue lengths as load indices" Proc. USENIX Winter Conf., Washington, DC, Jan. 1987, pp. 73-82.

[ZHOU88] S. Zhou. "A trace-driven simulation study of dynamic load balancing" IEEE Trans. Software Engineering, vol. 14, no. 9, pp. 1327-1341, Sept. 1988.

[ZUÑIGA] Fabiel Zúñiga G. "Suitable Behaviors in Dynamic Virtual Environments" PhD thesis, June 2007



CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

"2010, Año de la Patria, Bicentenario del Inicio de la Independencia
y Centenario del Inicio de la Revolución"

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Middleware para Distribución Adaptativa de Ambientes Multiagente
- Middleware for Adaptive Multiagent Environment Distribution

del (la) C.

Luis Alberto MUÑOZ GÓMEZ

el día 29 de Noviembre de 2010.

Dr. Luis Ernesto López Mellado
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara

Dr. Félix Francisco Ramos Corchado
Investigador CINVESTAV 3A
CINVESTAV Unidad Guadalajara

Dr. Mario Angel Siller González
Pico
Investigador CINVESTAV 2A
CINVESTAV Unidad Guadalajara

CINVESTAV - IPN
Biblioteca Central

SSIT0010073