

A MIDDLEWARE FOR UBIQUITOUS AGENTS

By Nikolaos Dipsis

*Thesis submitted to the University of London
for the degree of Doctor of Philosophy.*



*Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX,
United Kingdom
April 2015*

Declaration

I Nikolaos Dipsis hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

A handwritten signature in black ink, appearing to be 'N. Dipsis', written over a horizontal line.

Date: April 2015

Supervisor: Professor Kostas Stathis

External Examiner: Professor Gregory O'Hare

Internal Examiner: Professor George Roussos

Abstract

There is a paucity of pervasive middleware that allow integrations between Artificial Intelligence (AI) of the kind supported by software agents and sensor/actuator networks (SAN) while simultaneously featuring all of the following characteristics: a) a systematic way to achieve the integrations, b) transparency, c) multiple MAS (Multi-Agent System) support and d) sensor, actuator and device heterogeneity. The thesis is a step forward to developing pervasive middleware for AmI (Ambient Intelligence) by presenting an approach with all the above characteristics. We propose a middleware that creates *ubiquitous agents (UAs)* by embedding software agents in the physical world as part of a ubiquitous computing environment. We use intelligent software agents residing in a multi-agent system (MAS) environment. *UAs* are built through linking the functionality of agents residing in MAS networked environments, to aggregations of sensors, actuators and devices in the physical world that we call avatars.

The software agents consume services provided by physical sensors, actuators and SAN. The provider-consumer relationships enable agent functionality to access the data that is sensed by physical sensors and to also create effects in the physical world via physical actuators. Computationally expensive capabilities such as decision making and communication are performed by the agents in MAS platforms while the acting and the sensing in the physical world through their corresponding avatars.

Our approach follows SOA principles to implement a message oriented middleware that architecturally consists of: a) a base-layer enabling the sensors and the actuators to register as service providers and the agents to register as service consumers using an API (Application Programming Interface) and b) a reflection layer that creates models of agents and avatars using registration metadata from the base-layer and uses these models to create and manage *UA* functionality.

Furthermore, the *UA* framework uses the Z-notation for a detailed specification of every component of the middleware enabling researchers to implement it using different technologies. The eVATAR middleware that was developed in this thesis is such an implementation. eVATAR was applied on two scenarios for smart homes that use a MAS platform. Firstly we applied agent AI from the GOLEM and the JADE MAS on a miniature smart home containing real sensors and actuators in order to provide with evidence that the proposed approach is systematic, transparent and supports device heterogeneity and multiple MAS. Then we used a custom smart home simulation to illustrate the potential of a system using MAS agents, eVATAR and a sensor/actuator network embedded in a home context for becoming useful in confronting everyday lives problems. The thesis also includes a performance evaluation of eVATAR and discusses latency and how to reduce it. As future work we explore ways to improve our approach and to extend the scope of supported devices.

This thesis is dedicated to my family and especially to my sons Λεωνίδα and Αλέξανδρο.

Acknowledgments

Firstly, I would like thank my supervisor Prof. Kostas Stathis. Without his guidance, help, support, knowledge and constructive input this thesis would not have been possible. Also I would like to thank my family for their support and patience all these years.

Nikolaos Dipsis
April 2015



Table of Contents

1. INTRODUCTION	19
1.1. Motivation	20
1.2. Hypothesis.....	22
1.3. Aims and Objectives	23
1.4. Approach	24
1.5. Contribution.....	27
1.6. Structure.....	28
1.7. Previous publications.....	29
2. Background	31
2.1. Middleware for Distributed Applications	31
2.1.1. Categorization of Middleware	33
2.1.2. Middleware Approaches of Interest.....	40
2.2. SAN Middleware Encompassing Intelligence	42
2.2.1. SensorWare	42
2.2.2. Agilla	43
2.2.3. Impala	44
2.2.4. Discussion.....	45
2.3. SAN Middleware Supporting the Integration with External Applications.....	45
2.3.1. Open Geospatial Consortium (OGC) Sensor Web - NOSA	46
2.3.2. OASiS	47
2.3.3. TinySOA	47
2.3.4. USEME.....	48
2.3.5. SIXTH.....	48

2.3.6.	Discussion.....	49
2.4.	Pervasive and Robotics Middleware	49
2.4.1.	SALSA.....	49
2.4.2.	Amigo.....	50
2.4.3.	RoboCare	51
2.4.4.	iCore	52
2.4.5.	PEIS	53
2.4.6.	The “Middle Layer for Incorporations”	53
2.4.7.	MARIE	54
2.4.8.	The Player/Stage project.....	55
2.4.9.	ReMMoc.....	56
2.4.10.	Dynamic-TAO & CARISMA.....	57
2.4.11.	Discussion.....	57
2.5.	Evaluation.....	58
2.6.	Summary.....	61
3.	Requirements & Architecture.....	62
3.1.	Overview	62
3.2.	MAS and Physical World	69
3.2.1.	MAS Scope	69
3.2.2.	Physical World Scope	72
3.3.	The Middleware Requirements.....	73
3.3.1.	The Life-Cycle of a UA	74
3.3.2.	Registration	75
3.3.3.	Discovery and Binding.....	78
3.3.4.	UA sessions	80

3.4.	The Middleware Architecture	81
3.5.	Framework Specification.....	85
3.6.	Summary.....	90
4.	The UBIQUITOUS AGENTS Framework.....	91
4.1.	Specifying the Middleware	91
4.1.1.	Middleware definition.....	92
4.1.2.	Interactors as Service Providers and Consumers	94
4.1.3.	Base Layer Functionality	95
4.1.4.	The Entity Models in the Reflection Layer	96
4.1.5.	The Interactor Models in the Reflection Layer.....	99
4.1.6.	Registration	101
4.1.7.	Reflective Discovery & Binding	102
4.1.8.	Ubiquitous Agents	107
4.2.	Specifying the Middleware API	108
4.2.1.	API Definition.....	109
4.2.2.	Registration	111
4.2.3.	Messaging.....	113
4.3.	Summary.....	116
5.	eVATAR.....	117
5.1.	The Base Layer of eVATAR	117
5.1.1.	The BIL Metadata Language.....	118
5.1.2.	Base Layer Components.....	124
5.1.3.	eVATAR API.....	128
5.2.	The Reflection-layer of eVATAR.....	130
5.2.1.	Models.....	130

5.2.2.	Reflection.....	132
5.3.	Summary.....	137
6.	Case Studies	139
6.1.	Miniature RoboHome	139
6.1.1.	Scenario 1 – A Security Scenario using GOLEM	140
6.1.2.	Scenario 2 – A Simple Scenario Using JADE.....	150
6.1.3.	Discussion.....	156
6.2.	Simulated RoboHome.....	158
6.2.1.	The Scenario	159
6.2.2.	The RoboHome Simulation	160
6.2.3.	The Implementation of the Scenario.....	165
6.2.4.	Discussion.....	167
6.3.	Summary.....	168
7.	Evaluation & Discussion	169
7.1.	Performance & Scalability Testing.....	169
7.1.1.	Latency in eVATAR.....	169
7.1.2.	Test	171
7.1.3.	Results.....	172
7.1.4.	Discussion on Performance	175
7.2.	Failure Handling & Resilience.....	177
7.2.1.	Failure at Agent & Avatar level	177
7.2.2.	Failure at eVATAR level & Continuity	178
7.3.	Comparing eVATAR against related work	179
7.4.	Summary - Conclusion	182
8.	Epilogue.....	183

8.1. Synopsis	183
8.2. Future Work	185
8.2.1. Performance vs Scalability	185
8.2.2. Continuity of Service	186
8.2.3. Expanding the Scope of Supported Devices	186
8.2.4. Deployment - Usability	188
8.2.5. From BIL to a SensorML ontology.....	188
8.2.6. Security	189
8.2.7. Enabling Robots To Access Virtual Reality	190
Appendix A.....	191
A-1	191
A-2	192
A-3	193
Appendix B	195
B-1.....	195
Schema 1: XML Schema Definition (XSD) for BIL descriptions	195
Schema 2: XML Schema Definition for BIL messages.....	196
B-2.....	197
Child Elements of "Service"	197
B-3.....	199
eVATAR Base Layer UML classes	199
eVATAR API UML classes	209
Models Package UML classes.....	212
Reflection functionality package UML classes	214
Appendix C.....	222

C-1.....	222
C-2.....	223
C-3.....	225
C-4.....	226
C-5.....	227
C-6.....	227
Appendix D.....	229
D-1	229
References.....	230

List of Figures

Figure 2-1. Middleware in the OSI reference model.	32
Figure 2-2. MOM based on a “store and forward” approach. Sending applications place their messages in local message queues (should they exist) while receiving applications retrieve them from their own local message queues (again if they exist). The integration brokers store the messages from sender applications in queues and forward them from the queues to the appropriate recipient applications. The purpose of the integration brokers is to decouple client and server applications ensuring asynchronous interaction (only loose coupling is supported).....	34
Figure 2-3. Publish-subscribe MOM ([70], [37]). The publisher applications produce some type of information in the form of events (messages) and publish it via the integration broker. The subscriber applications express interest in these events by subscribing to receive them. The integration broker receives the published events and notifies the applications that have subscribed to them. The integration broker matches events against subscriptions. The published events are placed in a queue for each subscriber.	34
Figure 2-4. Reflective middleware ([63], [18]).	36
Figure 2-5. Feature tree for a taxonomy of middleware characteristics.....	37
Figure 2-6. Roles and interactions in SOA.....	39
Figure 2-7. Tree depicting the middleware approaches of interest. In accordance to our selection criteria this tree only includes middleware that can support intelligence. Also, we tried to include approaches with high impact on their domains. Note that a middleware may belong to more than one category.	42
Figure 3-1. We see an overview of the proposed system architecture. In our approach, any external data source that uses the API is considered to be a sensor. Similarly, any external to the middleware resource that can produce an effect in the physical world and uses the API is considered to be an actuator. Accordingly, our approach considers agent sensors as code that internalizes sensory data to the agent mind/reasoning component from the API and	

actuators as code that enables this component to interact with the API in order to create control messages (section 1.4). The technologies referenced in the above figure are exemplifying what could potentially use the API to interact with the middleware. In 3.2.2 we describe how these technologies use the API.63

Figure 3-2. The architecture of an agent in GOLEM (borrowed from [82]). Interaction in the platform is event-based. Actions that happen in the MAS environment notify the body sensors. Similarly, actions of the agent via the body's actuators generate events represented as attempts of action in the MAS environment. Such attempts happen if the action is possible.70

Figure 3-3. Middleware for Ubiquitous Agents.....74

Figure 3-4. The life-cycle of a Ubiquitous Agent.75

Figure 3-5. Example of a Ubiquitous Agent session.80

Figure 3-6. The architecture of the proposed middleware.....82

Figure 5-1. BIL description of an actuator.120

Figure 5-2. BIL snippet from the BIL description schema of Appendix B-1.....120

Figure 5-3. BIL Message.123

Figure 5-4. Example of child elements in the "Service" tag.....123

Figure 5-5. Sample code describing the functionality of a "connectionLoop" object.126

Figure 5-6. Sample code describing the mediation within a Ubiquitous Agent.136

Figure 6-1. The architecture of the system that is described in this scenario. ...141

Figure 6-2. BIL Description of the keyboard sensor in the miniature smart home.143

Figure 6-3. BIL Description of the motion detection sensor in the miniature smart home.....151

Figure 6-4. BIL Description of the motion detection sensor of the JADE agent.153

Figure 6-5. The sensing behaviour of the JADE agent.154

Figure 6-6. The acting behaviour of the JADE agent.....155

Figure 6-7. The RoboHome architecture reference model.160

Figure 6-8. Subclasses of the “Device” superclass.....	162
Figure 6-9. Electricity Consumption with and without intervention from UAs. The time in seconds refers to the scaled real time.	166
Figure 6-10. Costs with and without intervention from UAs.	167
Figure 7-1. Using Equation 1 to create an average latency over time graph for the first piece of hardware. Appendix D-1 shows the numbers of agents, avatars and interactors that are being served by eVATAR as they increase during the execution of the test.	173
Figure 7-2. Bell curve of the distribution of the mean latencies. The mean here is the mean of the mean latencies. We see that most values are close to the mean.	174
Figure B-0-1. BIL interactor description XSD.	196
Figure B-0-2. BIL message XSD.	197
Figure C-0-1. Calculation of electricity consumption and related costs for a device.	225

List of Tables

Table 2-1. Cross-reference of approaches and middleware characteristics of middleware that support AI by using agents as part of their implementation or via integration.	58
Table 2-2. Evaluation of pervasive/ AmI middleware in relation to the aims of the thesis.	60
Table 3-1. Z-notation sample reference table.	89
Table 5-1. The child elements of the “bilinfo” element that describe an interactor.	119
Table 5-2. The child elements of the “Service” element that describe a service.	121
Table 6-1. “Miniature Smart Home” sensors and actuators.	142
Table 6-2. The GOLEM agent interactors.	146
Table 6-3. Avatar interactors for the second scenario.	150
Table 6-4. The JADE agent interactors.	152
Table 7-1. Test results with averages of latency means for the 50 runs of each test and the connection thresholds for the different hardware settings.	175
Table 7-2. Evaluation of pervasive/ AmI middleware and eVATAR in relation to the aims of the thesis.	180
Table 7-3. Cross-reference of approaches and technologies used by the different middleware.	180
Table B-0-1. Describing the child elements of the “Service” tag for the BIL message.	198

Z-Notation Listings

Listing 4-1. The state space of the middleware system.	93
Listing 4-2. Creating global variables for sets that describe elements of the middleware.	93
Listing 4-3. Incoming metadata description from an interactor.....	94
Listing 4-4. Base layer functions.....	95
Listing 4-5. Specifying the receiving and processing of registration data by the middleware.	96
Listing 4-6. Reflection layer functions for the creation of entity models.	97
Listing 4-7. Creating the entity model metadata.	97
Listing 4-8. Updating the metadata of an entity model.....	98
Listing 4-9. Return the entity model and create one if it does not exist.	99
Listing 4-10. Adding an interactor model to an entity model.	99
Listing 4-11. Reflection layer functions for the creation of interactor models. ...	99
Listing 4-12. Creating a new interactor model while also adding the metadata.	100
Listing 4-13. The incoming and outgoing queues for the particular interactor identifier.	100
Listing 4-14. This function specifies the addition of a new interactor model to an entity model.	101
Listing 4-15. Interactor registration	102
Listing 4-16. Check whether to attempt a binding.	103
Listing 4-17. Support function for binding.....	103
Listing 4-18. Binding two interactors.	103
Listing 4-19. A relationship between an entity model and a UA session.	104
Listing 4-20. Creation of a UA.....	104
Listing 4-21. Evaluates two entity bodies and binds them.	106
Listing 4-22. The Bind operation.....	106
Listing 4-23. Message mediation within a UA.....	108

Listing 4-24. The schema of the middleware API.....	110
Listing 4-25. An “API” object for accessing the variables of the API schema. ..	111
<i>Listing 4-26. Adding metadata to a middleware message.</i>	<i>111</i>
Listing 4-27. API functions for acquiring metadata that describes the interactor.	111
Listing 4-28. Getting the metadata that describes the interactor.....	112
Listing 4-29. The registration of the DIS.	113
Listing 4-30. This API function uses the service description element of the metadata description to create messaging metadata for describing sensing/acting requests or action feedback/sensory data messages.....	113
Listing 4-31. Adding input data values describing interactor activity to the metadata of a sensing/acting message.	114
Listing 4-32. The DIS uses the message metadata that it has acquired from the “apiGetMessageMetadata” function and adds to it at runtime the parameterized data values describing a sensing/acting request or an action feedback/sensory data message.	114
Listing 4-33. Operation that is used by DIS for sending messages to the middleware. The DIS creates the messages using the “apiEditMessageMetadata” function.	115
Listing 4-34. The operation for receiving a message from the middleware and returns the metadata of the message.	115
Listing 4-35. Specifies the API operation for enabling the DIS to extract the values that describe a sensing/action request or sensory data/feedback. The message metadata is acquired from a received message using the “apiReceiveMessage” operation.	115

1. INTRODUCTION

We live in a world in which people are increasingly relying on electronic devices in order to carry out their everyday activities. More and more of these devices can connect to networks like the Internet and can be used as sensors or actuators, depending on their purpose in the environment in which they are situated ([38], [79]). It is anticipated that the numbers of networked devices and sensors will reach to the levels of tens of billions and trillions respectively [37].

The abundance of connected devices and sensors offers a great opportunity for realizing computing concepts such as the vision for Ambient Intelligence (AmI) that aspires to utilize them for integrating services in our environments at home, at work, on the move, for entertainment, transport, healthcare and in general in most environments inhabited by people. AmI is a relatively new paradigm in computing referring to digital environments that are sensitive, responsive and potentially adaptive to human needs and behaviours aiming to transparently empower human capability ([18], [29]). We are particularly interested in domestic AmI applications (often referred to as smart homes).

AmI combines elements of Ubiquitous Computing, Artificial Intelligence (AI) and sensor/actuator networks. It has its roots in Ubiquitous Computing which was described by Mark Weiser [60] as the concept of computers “weaving themselves into the fabric of everyday life until they are indistinguishable from it.” AmI systems use contextual information provided by the embedded in the environment sensors to adapt it in a way that adds to human capability in a transparent, unobtrusive manner. Other characteristics of AmI are personalization, being anticipatory and being adaptive to the needs of individual users and to changes in the environment [32].

Several characteristics and especially the intelligence aspect of AmI are typically drawn from AI techniques. AI techniques offer great potential for adding further sophistication and intelligence to the processing of sensory data in

sensor and actuator networks (SAN) [40]. Furthermore, the reasoning and adaptability of AI agent architectures that are typically designed for dynamic environments (see [30] and [51]) can implement AmI when applied to SAN.

Software agents often support high-level software capabilities, such as reasoning and planning, because they usually run on computationally powerful and network connected MAS platforms, in contrast to the low-level capabilities of sensors and actuators that are commonly used in smart home SANs. By integrating the two, the SAN can benefit from the computation and communication capabilities of the MAS platform. This way agent technology can be used to make devices in a SAN smarter in a cost effective way without requiring costly hardware upgrades for implementing more intelligent behaviours.

In the thesis we investigate how to create a systematic way for enabling developers to integrate software agent functionality into networks of sensors and actuators for the creation of AmI in a smart home. We want to provide a framework that is transparent to the developers concealing the low-level implementation details that describe how the integrations are achieved. This way we provide developers with a solution to the problem of achieving such integrations when implementing AmI systems that also require the use of AI of the kind supported by software agents. The framework should also allow developers to use a MAS platform of their choice based on their application requirements, MAS platform availability and personal preferences (i.e. they should not need to be trained to program a new MAS platform if they are already familiar with one). Furthermore, due to the aforementioned anticipated increase in the numbers of sensors and connected devices in our environments, the framework should be able to manage heterogeneity.

1.1. Motivation

The inclusion of software agent AI functionality in wired or wireless SAN/SN (Sensor Networks) is usually implemented by middleware. There are numerous

middleware based approaches that could potentially be used for integrating software agent intelligence with sensor and actuator networks. We can identify three categories: agent-based SAN/WSN (Wireless Sensor Networks) middleware, SAN/WSN middleware that support integration with external resources and applications (e.g. a MAS) and high-level middleware for pervasive systems. The first two categories describe middleware approaches that implement SAN or WSN. The latter describes middleware approaches that use sensory data directly from sensors or as derived via a SAN/WSN middleware in order to support pervasive computing applications.

Agent based middleware frameworks for WSN such as [8], [91] and [17] are usually implemented with an intrinsic support of a single platform overlooking heterogeneity issues. Also, such middleware approaches are usually bound to a single agent platform. Furthermore, it is a challenging task for developers to program agents in decentralised nodes of a SAN to coordinate their activities and perform sophisticated cooperative tasks.

The second category of SAN middleware of interest include SOA (Service Oriented Architecture) based implementations that besides providing with a way to interconnect heterogeneous SAN/WSN nodes, they also enable them to interact with external applications such as a MAS. Middleware approaches of this category include [62], [10], [23] and [95]. SOA based approaches can generally implement the communication between SAN and an external to the SAN application which in our case would be a MAS. We are interested in systematizing a specific application that deals with the dynamic creation, management and monitoring of relationships between software agents and hardware sensors/actuators at runtime. Middleware approaches of this category cope well with heterogeneity in terms of physical devices and supported MAS platforms but a higher level middleware would be required to systematize how such integrations would be achieved in a transparent way for the system developers.

The aforementioned SOA based approaches are still dependent on the low-level SAN / WSN functionality and they tend to focus on gathering information from sensors while a higher-level middleware would focus on using this information efficiently. There is a variety of such middleware approaches from the areas of pervasive computing e.g. [43], [105], [81] and [108]. Similarly to the low-level SAN middleware approaches, there is a paucity of pervasive middleware that allow integrations between software agents and SAN while simultaneously featuring all of the following characteristics:

1. A systematic way to achieve the integrations.
2. Transparency.
3. Multiple MAS support.
4. Sensor, actuator and device heterogeneity.

To support this claim in the Background chapter 2 we describe in more detail and evaluate the above categories of middleware and approaches as well as a number of other middleware frameworks that were carefully selected based on their potential to achieve the integration of agent AI with sensor and actuator networks and their impact on their respective areas of application.

1.2. Hypothesis

Our hypothesis is that it is possible to create a middleware framework for enabling developers to integrate software agent functionality into networks of sensors and actuators for the creation of AmI in a smart home that is simultaneously:

- Systematic.
- Transparent to the developers.
- Supporting multiple MAS.
- Supporting heterogeneous devices and software.

1.3. Aims and Objectives

In this section we make explicit what the aims and objectives of the thesis are and how they will be demonstrated. The aims of the thesis are as follows:

1. Create a middleware framework that can integrate software agent functionality with a SAN. This will be demonstrated by providing with a proof of concept in which the framework enables a software agent to interact with a set of physical sensors and actuators.
2. Develop a systematic way in the framework to integrate software agent functionality with sensors and actuators. It should include a standard way for implementing the integrations. This will be demonstrated by implementing at least two case studies each following the exact same method to enable software agents to be applied to SAN.
3. Support system developers who will act as the users of the framework to transparently integrate software agents with sensors and actuators. The developers should include the middleware functionality in their applications using abstract interfaces that conceal the low-level details of how software agents are applied to a SAN. Transparency should be demonstrated by providing case studies that implement the integrations using abstract interfaces without the requirement to write application code for dealing with how the integrations are achieved.
4. Provide framework support for multiple MAS platforms. This will be demonstrated by creating at least two case studies each providing a proof of concept using a different MAS platform.
5. Offer framework support for heterogeneous sensors, actuators and devices. This will be demonstrated by creating case studies that involve a variety of heterogeneous devices.

In order to achieve the above aims, the objectives of this thesis are as follows:

- I. Use the Z-notation [45] to specify a framework for a middleware that fulfils the aims of the thesis as stated above.
- II. Implement the middleware specified in the framework.
- III. Evaluate the proposed approach as described in the Z-Notation framework and its implementation in terms of satisfying the aims of the thesis using an application that integrates the GOLEM MAS [82] and real sensors and actuators and an application using JADE [25].
- IV. Implement a test suit consisting of a smart home simulation and a MAS for simulating smart home scenarios that involve the integration of MAS AI to smart home sensor/actuator networks and implement a scenario that shows in simulation the potential of our approach in becoming useful in a person's every day activities.
- V. Use the simulation to test the middleware's performance.
- VI. Compare the proposed approach with other approaches in terms of achieving the aims of the thesis.

1.4. Approach

In the same spirit as Mark Weiser's [60] definition of ubiquitous computing this thesis presents an approach for ubiquitous agents (UAs) by "weaving software agents into the fabric of everyday life until they are indistinguishable from it." At this point it would be useful to provide a brief description of the software agents of the approach. Shoham in [96] defines agents as software entities functioning in a continuous and autonomous manner within a specific environment that is potentially also a home to other agents, software processes and objects. Within the context of this work and based on the definitions of Franklin and Graesser in [85], we will be calling agents those software programs that have the following features: reaction to the environment, autonomy, goal-orientation and persistence. According to Wooldridge and

Jennings in [61], agents also present social ability enabling them to communicate between them.

Agents are being reactive by responding to perceived changes in the agent environment. Persistence refers to agent continuity within the agent environment. According to [42], agents need to satisfy the requirements for continuity and autonomy in order to be able carry out activities in the environment without the need for constant guidance or intervention by other entities such as human users or software programs. A goal directed behaviour enables purposeful action to the reactive, autonomous and persistent nature of agents.

It would be useful to introduce the notions of the software agent “mind” and “body” as well, which are integral to the proposed in this thesis approach as we will see in the following. More specifically, the separation of the “mind” (supporting capabilities such as decision making and reasoning) from the agent’s “body” (responsible for aggregating the sensors/actuators situating the agent on a distributed application environment) has been for a long time common practice both in the MAS literature [52], but also in MAS development practice ([82], [50], [84] , [4]). Thus, at least in principle, it should be possible to link an agent’s software “body” to one with presence in the physical environment.

In the rest of the thesis the notion of “Ubiquitous Agent” or UA will be used to describe an entity that is capable of simultaneous presence in both a MAS electronic and a physical environment. UAs are built through enabling software agents residing in electronic networked environments, to connect to and control the aggregation of one or more devices in the physical world. At the same time their bodies may still bare sensors and actuators allowing them to interact with their electronic MAS environment.

Furthermore, the term avatar will represent any aggregation of connected devices whether sensors, actuators or smart devices that are embedded in a finite physical environment such as a smart home (including mobile robots).

Our use of the term “avatar” does not follow the popular computing definition for an avatar being a graphical representation of a user [99]; instead we defined it to be the software agent’s bodily form in the physical environment via a set of physical sensors and actuators.

Therefore, a UA = Software Agent + Avatar. It is assumed that the software agents usually run on powerful, in terms of computational capabilities and connectivity, MAS platforms. The computationally expensive capabilities such as reasoning, planning and communication are performed by the agents in the MAS platform while the acting and the sensing in the physical world through the corresponding avatars. The UAs are viewed as connected within a network as all agents within a MAS such as JADE [25] or GOLEM [82] are capable of communicating with each-other within the MAS environment therefore the same applies for the UAs they belong to. This way, even the simpler hardware of the sensor-actuator network appears to have access to more information and implement more context aware behaviours when powered by MAS agents. An Aml system in the context of this architecture is typically a network of nodes that are UAs.

In our approach, sensors and actuators interact with the middleware using an API (Application Programming Interface). The API provides an XML based metadata language providing an ontology that enables abstract interaction with the middleware in order to create agent – SAN integrations concealing the low-level details of how these integrations are achieved (transparency). In our approach, any external data source that uses the API is considered to be a sensor. Similarly, any external to the middleware resource that can produce an effect in the physical world and uses the API is considered to be an actuator. An avatar body is a collection of such sensors and actuators. Accordingly, our approach considers agent sensors as code that internalizes sensory data to the agent mind/reasoning component from the API and actuators as code that enables this component to interact with the API in order to create control

messages. In our approach we call an agent body a collection of agent sensors and actuators as described above.

Our approach follows Service Oriented Architecture (SOA [57] and [89]) principles to implement a message oriented middleware that architecturally consists of: a) a base-layer enabling the sensors and the actuators to register as service providers and the agents to register as service consumers using the API and b) a reflection layer (see [33] and [27]) that creates models of agents and avatars using registration metadata from the base-layer and uses these models to create (dynamic discovery) and manage UA functionality (mediator). The provider-consumer relationships enable software agents to access the data that is sensed by sensors and to also create effects in the physical world via actuators.

In the thesis, sensors and actuators are also referred to as “interactors” as they are the means by which an agent may interact with the physical environment via sensing and acting. We use the term “interactor” for both physical sensors/actuators as well as the sensing and acting software components of a software agent body.

1.5. Contribution

This work contributes an approach consisting of an architecture and a framework for a middleware component providing a link between MAS and SAN networks using UAs. The Z notation is used to specify the components of the framework and their interaction. The Z specification of the framework aims to describe a systematic and transparent way for linking software agents to sets of physical sensors and actuators that is implementation independent, thus allowing different researchers to implement it with different technologies. Also the framework is MAS platform and physical device platform independent.

Another contribution of the work is the implementation of a middleware platform according to the specified framework. This platform consists of a) the

eVATAR middleware (a play with the words electronic and avatar to denote that the middleware enables an electronic entity i.e. a software agent to have an avatar in the physical environment) and b) BIL (for Body Integration Language), a metadata language for describing sensors and actuators belonging to agents or embedded in physical environments. EVATAR implements a SOA to provide with systematic way for linking agent and SAN functionality. It also features a component that implements reflection allowing it to dynamically link agent bodies to physical avatars and to manage and monitor their interactions and the middleware itself. The thesis also contributes a proof of concept involving the implementation of a smart home security scenario using EVATAR, real sensors/actuators embedded in a miniature house and the GOLEM MAS [82]. Also, a simpler application using a JADE agent [25] was implemented indicating MAS platform independence. A simulation of a smart home that we call RoboHome is another contribution of this work. It was used with EVATAR and the GOLEM MAS to evaluate EVATAR in terms of improving everyday life activities. In the case studies we used different hardware and software and implemented different applications following the same systematic method (evidence with regards to the systematic aim). Furthermore, we used high level interfaces to interact with EVATAR (transparency aim). Finally, the scenarios included heterogeneous devices, this way satisfying the relevant aim.

The experience gained from the implementation of these scenarios as well as the results of the performance testing of EVATAR were used as the foundation for the final contribution of this work which is summarized in an evaluation and a discussion regarding EVATAR, UAs and AmI. This work also provides with ideas regarding future work.

1.6. Structure

After this introductory chapter, chapter 2 is describing the background for the thesis presenting the state of the art as well as the concepts and influences of the

described work. Chapter 3 presents the proposed architecture and chapter 4 formally describes the framework of the proposed approach. In Chapter 4 in particular, the Z-Notation is used to specify a middleware that can be used for creating UAs. Chapter 4 also specifies an API enabling MAS and physical sensors/actuators to participate in an AmI system that uses UAs.

Chapter 5 follows, describing EVATAR which is essentially an implementation of the framework using SOA principles and reflection. In chapter 6, we can see the framework and EVATAR being utilized in two case studies. In the case studies we see how we achieved the aims that were set in the introduction and also collect data that will be used in the evaluation/discussion chapter that follows. Finally, in the evaluation/discussion chapter 7 we see an evaluation of EVATAR with a specific focus on performance and scalability. This chapter also discusses failure handling in systems that use EVATAR. It concludes with a discussion on related work, aiming to evaluate EVATAR against other middleware. The thesis concludes with an assessment epilogue (chapter 8) and some concrete ideas and considerations for future work including a discussion regarding the potential of using EVATAR and MAS AI to create an infrastructure for the “Internet of Things” (IoT) [37], [70].

1.7. Previous publications

The thought process that has led to this thesis can be reflected in a number of publications. The beginning of this process was signified by the conception of the idea for devising a systematic way for approaching the task of amalgamating software agents with physical environments while considering issues that deal with action and perception in [67]. The first version of eVATAR was presented shortly after in [68] where we introduced the basic ideas behind it and presented it from a Service Oriented Architecture (SOA [89], [58]) perspective. In [69] we see a more complete version of EVATAR featuring a more resilient approach and an architecture that fits in with AmI systems. This thesis presents the latest version of eVATAR that also features a metadata layer

for implementing reflection that enables eVATAR to dynamically link agent to avatar bodies and ensure service continuity and resilience. This thesis also provides with a framework and a more concrete approach for testing and evaluating the middleware.

2. BACKGROUND

Our domain of interest is predominantly smart home environments. Software integrations in networks of sensors and actuators such as the ones encountered in smart homes are commonly pursued with the use of middleware. In this chapter, we examine middleware approaches for such systems in relation to the task of enabling software agents to connect with networks of sensors and actuators in order to sense, act and implement intelligent behaviours in physical world environments. This chapter essentially focuses on the different types of middleware, also identifying concepts and relevant approaches and technologies to the work described in the thesis. At this point we can proceed with defining what middleware is and identify the particular category of middleware that are relevant to our problem.

2.1. Middleware for Distributed Applications

In this section we describe what constitutes a middleware for distributed applications. The concept of middleware revolves around a piece of connectivity software enabling the interaction between software processes that are running on distributed within a network machines or on the same machine [57] (in this thesis we focus on middleware for distributed applications). Architecturally, middleware operates on a layer between the operating system (OS) and the distributed application layer. Figure 2-1 illustrates middleware in relation to the Open Systems Interconnections (OSI) model [39] for communication systems. In particular, middleware operates on the session and presentation layers. The first is responsible for the establishment, management and termination of connections between local and remote applications within the network. The presentation layer deals with the context and representation of data derived from application software that could potentially be heterogeneous.

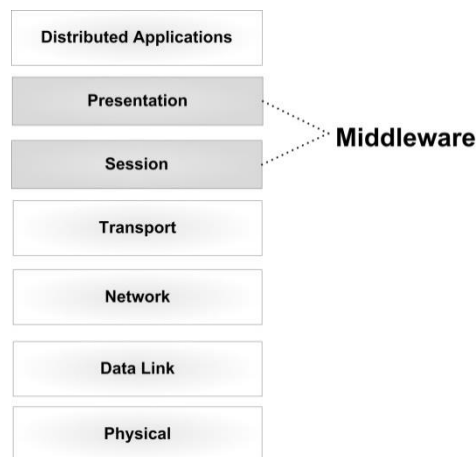


Figure 2-1. Middleware in the OSI reference model.

Middleware hides the complexity of distributed systems providing with an infrastructure for distributed applications in a way that enables communication among heterogeneous platforms, thus allowing for interoperability. It commonly resolves heterogeneity issues stemming from the use of a different operating systems, hardware and networks within the same distributed system.

According to [57], middleware provides interfaces in order to:

- Abstract distributed application functionality allowing developers to implement distributed systems without having knowledge of application implementation details.
- Provide platform transparency enabling distributed applications to interact with each other without the need to take into consideration of each other's underlying implementations.
- Provide location transparency enabling applications to interact with other applications across the network as if they were local to them and without having knowledge of their location.
- Offer services such as authentication and security with regards to the interaction of the distributed applications.

2.1.1. Categorization of Middleware

Middleware is often characterized by the model of interaction they implement. In synchronous communication, the execution flow of the caller code is suspended (blocking and waiting), until the called code in a distributed location is executed and returns control. A synchronous middleware (e.g. a middleware using RPC [57]) presents tight coupling meaning that the calling and the called code are highly dependent on one another. In asynchronous interaction, the caller code does not need to block and wait for the called code to return control and it can continue with its tasks. The called code does not need to execute immediately either. Hence, asynchronous interaction commonly promotes loose coupling meaning the calling and the called code have no knowledge of or dependencies to each other. This is very useful in environments with processes whose behaviour can vary depending on their design, runtime usage and the workload or time required to perform a task.

The need for pure asynchronous communication and loose coupling led to the development of the “Message Oriented Middleware” or MOM paradigm. The two main types of messaging that are used for asynchronous communication in MOM [57]: message queueing (also known as “store and forward” MOM) and publish-subscribe (also known as “event-based” MOM). The messages are typically described in XML format [107]. The architecture of the “store and forward” approach features client applications, server applications and a message server between them. Message servers are commonly referred to as “integration brokers”. An integration broker is a separate architectural element within the distributed environment (see Figure 2-2). Its purpose is to store the messages it receives from sending applications in message queues and forward them to the applications that receive them.

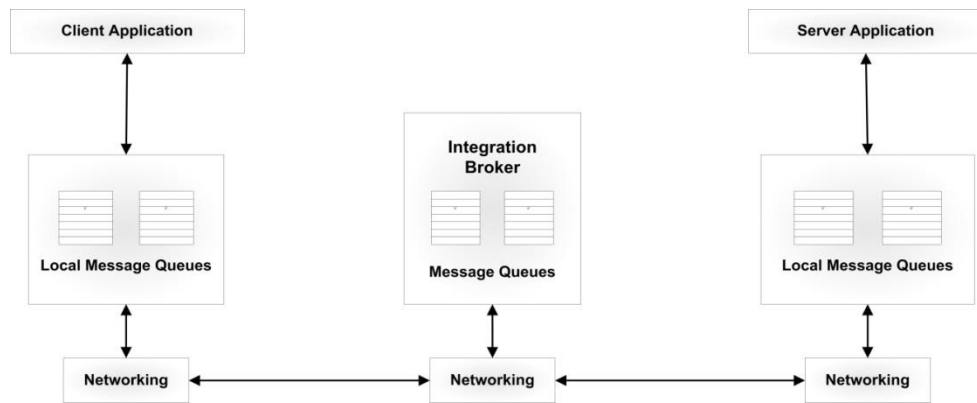


Figure 2-2. MOM based on a “store and forward” approach. Sending applications place their messages in local message queues (should they exist) while receiving applications retrieve them from their own local message queues (again if they exist). The integration brokers store the messages from sender applications in queues and forward them from the queues to the appropriate recipient applications. The purpose of the integration brokers is to decouple client and server applications ensuring asynchronous interaction (only loose coupling is supported).

The sending and receiving applications commonly use adapter ports that ensure that the required message format is used to interact with the integration brokers [57].

Publish-subscribe MOM middleware implement an architecture consisting of three main elements ([57], [71]): the publisher applications, the subscriber applications and the event service (integration broker).

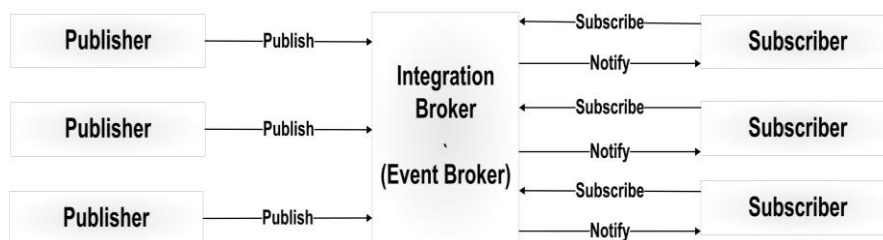


Figure 2-3. Publish-subscribe MOM ([57], [71]). The publisher applications produce some type of information in the form of events (messages) and publish it via the integration broker. The subscriber applications express interest in these events by subscribing to receive them. The integration broker receives the published events and notifies the applications that have subscribed to them. The integration broker matches events against subscriptions. The published events are placed in a queue for each subscriber.

The event notification service enables the asynchronous communication and loose coupling between heterogeneous clients. There are two types of subscriptions: topic (subject) based and content based [71]. In topic based subscriptions, the publishing applications publish events within specific topics (subjects) and the subscribers subscribe to particular topics of interest. In content based subscriptions, the publishers again publish events that belong to particular topics and the subscribers subscribe to them. The main difference is that these implementations filter which events are received by a subscriber from a topic that it has subscribed to [57].

Another common type of asynchronous communication uses shared tuples that can be accessed concurrently by processes in order to share data. We will see a number of examples of middleware that follow the message oriented approach (e.g. [43], [105] and [81]) and the tuple spaces approach (e.g. [17], [6] and [7]). The Object Oriented technologies such as RMI [112] or CORBA [86] would not be very suitable for dynamic environments such as ones containing sensor actuator networks. The first provides synchronous communications which as we saw previously is not suitable for the dynamic environments containing sensor and actuator networks and the latter promotes asynchronous communications but does not strictly present loose coupling due to the lack of a versioning mechanism [86].

Another important characteristic deals with the ability of a system to reason about itself enabling it to inspect and change itself during runtime. The reflective middleware model was conceived for dynamic environments in order to support the development of flexible and adaptive distributed systems [27]. The foundations for reflection were laid out in [13] and a system is reflective if it reasons about itself and therefore it is also able to inspect and change itself during runtime. There are two levels in a reflective application, the base-level and an internal representation/model of the base-level that is called the reflection-layer. In object-oriented reflective systems, the entities populating the reflection-layer are called meta-objects and they reflect base-level objects

(Figure 2-4). The base-level deals with the usual functionality of the application within its problem domain while the reflection-layer reasons about the base-level in order to be able to enrich, monitor and potentially change the behaviour of the application during runtime (Figure 2-4).

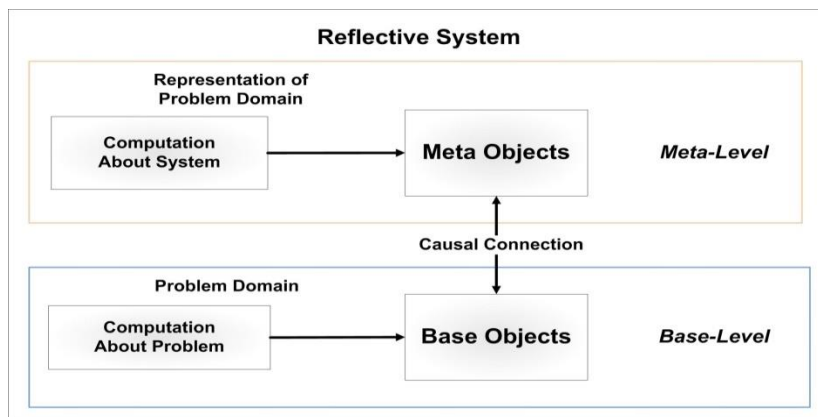


Figure 2-4. Reflective middleware ([27], [33]).

The fundamental concepts of reflective middleware ([27], [33]) are:

- Reification: the action of making aspects of the internal representation (meta-objects) of the middleware explicit and accessible from the base-level application.
- Absorption: the reverse of reification. It essentially consists of effecting the changes made to reified base-level objects into the internal representation of the system.
- Reification and absorption are realizing a causal connection link between the model and the application it models.
- A reflective system has a base-layer dealing with the application and a reflection-layer dealing with reflective computation.

Reflection has been used by middleware in order enable service discovery protocols to cope with heterogeneity with regards to connected services. It is commonly used to provide a second layer in the middleware on top of the heterogeneous service discovery protocols in order to map service discovery

queries from a particular protocol to service discovery queries of other potentially heterogeneous protocols.

The question here would be what other characterizations exist and is there a useful taxonomy that would classify existing approaches enabling us to find a solution to the problem of systematically linking MAS AI to sensor and actuator networks.

Focusing on our domain of interest, the latest advances in sensor networks and in particular in WSN (Wireless Sensor Networks) provide with a variety of middleware that could be used as a starting point for the discussion about the composition of our own solution. A number of comprehensive middleware reviews have been published in the domain of WSN for example [53] by Römer et al. and [66] by Wang et al. They offer taxonomies that are useful as a means of categorization. Wang et al. in [66] provided with a taxonomy containing important characteristics of such middleware. The following feature tree is inspired by the one in [66]. It divides the characteristics into two types: characteristics related to developing systems using the middleware and implementation characteristics.

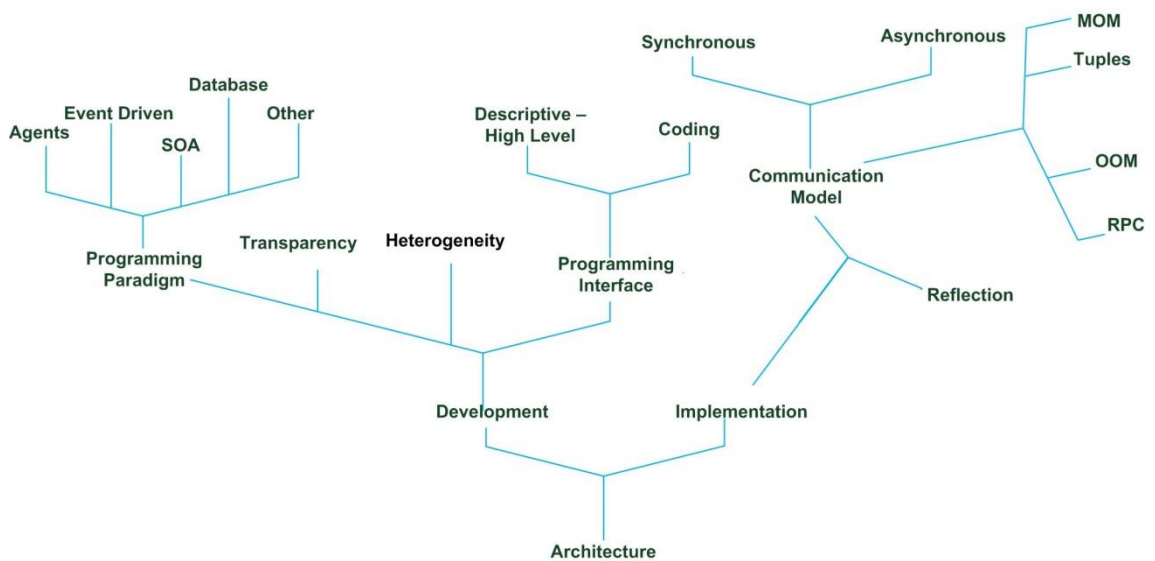


Figure 2-5. Feature tree for a taxonomy of middleware characteristics.

The programming paradigm refers to the structure and building components of the middleware. Common paradigms include agent based programming approaches where the agents implement the middleware functionality, database programming, event driven models (implementing the publish-subscribe [71] pattern) and SOA among others. In the following we will describe several middleware from each category. The SOA paradigm as we will see in the following middleware reviews has been used extensively for middleware integrations between heterogeneous platforms and we will describe it here in further detail.

According to Papazoglou ([58], [57]), Erl [89] and the OASIS (Organization for the Advancement of Structured Information Standards) [115] definitions, SOA is not a concrete architecture. It is instead a paradigm that leads to the implementation of concrete architectures for scalable distributed information systems featuring autonomous, interoperable, reusable, discoverable and multi-platform services. According to the same sources we can describe a service as a mechanism that provides interfaces for accessing one or more capabilities. A service can be viewed as an application that can be interacted with via a programmable interface. The main architectural elements of a SOA are the (see Figure 2-6):

- Service provider: the entity that uses the service to offer one or more capabilities.
- Service consumer: the entity that needs the capabilities offered by the service provider via the service. It is also known as the client.
- Service broker: an entity acting as a mediator. It offers directories that are used by service providers to publish their services. It also features functionality making the services visible and consequently discoverable by consumers.

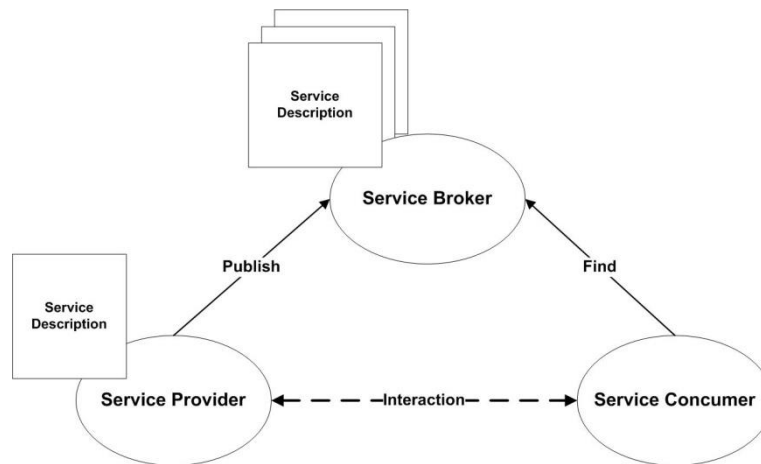


Figure 2-6. Roles and interactions in SOA.

The service providers advertise their services via the broker using service descriptions. The descriptions also define how to invoke and interact with a service using the service interface. The consumers search for desired capabilities of services in the directories of the broker according to their needs. Consumers use the information they acquire from the broker to bind to a required service and interact with it. In general the interaction between the services and their consumers is performed via messaging.

According to [58], [57], [89] and [115], a SOA is designed based on a number of principles as:

- Reusability: a service is implemented only once and can be used multiple times.
- Contract: the contract (see description in Figure 2-6) defines how to communicate/interact with the service.
- Discoverability: SOA should include a directory for enabling clients to determine which services have registered the capabilities they are after.
- Loose coupling: the calling and the called code have no knowledge of or dependencies to each other.
- Abstraction: the service interfaces conceal the implementation and the details of the provided by the service capabilities from the consumers of the service and the distributed system environment.

- **Composability:** ability to aggregate services to collectively automate a particular task or process. Services can also be used by other services. The service composition is an important concept that addresses the many aspects associated with combining services into aggregate distributed solutions. It deals with topics such as runtime messaging, message design and inter-service security controls ([89], [58]).
- **Autonomy:** Services have control over their underlying runtime execution environment and resources [89].
- **Statelessness:** state information between different service calls is not maintained and all temporary service information is discarded.

Based on the same sources (see [58], [57], [89] and [115]) interoperability is an intrinsic characteristic of SOA that follow the above principles.

The abstraction level in the feature tree of Figure 2-5 can be either low level i.e. programming each node individually or system level i.e. programming the system as a whole. In the second case, middleware add transparency by hiding the complexity of low-level functionality such as: a) code distribution, b) how the data is accessed from the nodes and c) how the nodes coordinate their tasks. Also, the programming interface defines how application developers interact with the middleware. This can be done with descriptive high-level interfaces (more resources) e.g. using XML or a type of syntax that resembles a database query (SQL query) or with imperative interfaces requires to write code to interact with the system (more complexity).

2.1.2. Middleware Approaches of Interest

In this section we justify the selection criteria for the middleware that have been chosen and reviewed in this chapter. We focus on the elements of the above taxonomy that are most relevant to the problem we are trying to solve. These elements are highlighted in the above tree structure and will be described and explained in relation to our problem as we go along with the chapter leading to a feature comparison of all the important technologies through a set of carefully

selected middleware that implement them. With regards to our problem, there are two ways that middleware commonly incorporate agent AI. The first is about middleware that purport to support intelligence, commonly by applying agents to each node in the sensor (and potentially actuator) network. These middleware implement the agent model programming paradigm (see Figure 2-5). The second way is by integration. According to [66] an important aspect of SN/SAN (Sensor Networks/ Sensor and Actuator Networks) middleware is the ability to facilitate the integration with internet, cloud based systems, database-based systems and external applications such as MAS.

Despite this, WSN middleware usually implement close networks i.e. networks that do not have connections with the outside world and therefore, another middleware would be required to enable such connections with external applications. High-level pervasive computing/AmI middleware can be used in conjunction with the SN/WSN middleware to implement integrations. According to [66] the increasing availability of heterogeneous sensors and actuators in our environments create new challenges in terms of dealing with complex heterogeneity problems. They argue that integrating WSN/SN middleware with high-level pervasive computing/AmI middleware is inevitable with the first focusing on gathering information from the physical world and the latter on using this information to support pervasive computing/AmI applications.

There is a plethora of middleware that are used in the WSN, SN and pervasive systems contexts. In the following we will review middleware approaches that enable the incorporation of AI in physical environments that are enhanced by sensor and actuator networks either directly by using agents or via integration to external applications in the internet, the cloud or a MAS. The following tree provides with an overview of such middleware.

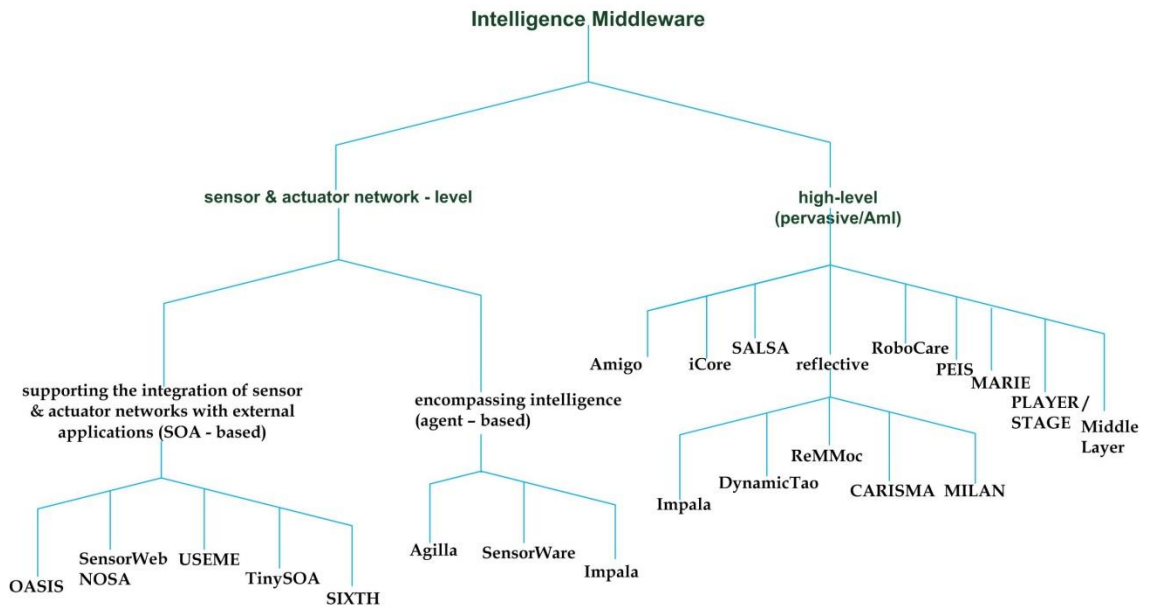


Figure 2-7. Tree depicting the middleware approaches of interest. In accordance to our selection criteria this tree only includes middleware that can support intelligence. Also, we tried to include approaches with high impact on their domains. Note that a middleware may belong to more than one category.

Also, our goal is to review middleware with high impact on their respective domains and in relation to the one of the thesis which is smart homes.

2.2. SAN Middleware Encompassing Intelligence

The types of middleware that we will be reviewing in this section already use software agents (in particular mobile agents) as part of their functionality. They create modular applications that distribute mobile code through the sensor network providing in this way dynamic reconfiguration capabilities and increased usability. They also react to their environment in ways that could potentially support intelligence.

2.2.1. SensorWare

The SensorWare [8] middleware is designed for application on networks of rich in terms of processing and memory resources devices (in particular on Wireless

Ad hoc Sensor Networks – WASN [9]). Nodes in such networks are described as providers of services that are consumed by agents. It allows service reconfiguration during runtime according to the needs of each application. The agents distribute the processing by executing tasks using services offered by nodes. Each node can facilitate a single or multiple agents that implement state machines. The state machine functionality enables the agents to respond to their environment, sensing events and timeouts. It follows the “active sensor” approach [8] abstracting the runtime environment of a sensor node by providing native services.

The system is programmed using a scripting language (TCL [46]) that also poses a requirement for expensive computation capabilities. TCL implements a high-level scripting language for programming the WASN allowing dynamic configuration and program updates. Furthermore, SensorWare enables the reconfiguration of nodes and the services that they offer at run time. For example in data fusion (aggregation) tasks, that could prove beneficial especially when changing the fusion (aggregation) rules and requirements at runtime.

The main drawback of this approach is that it requires significant processing, memory and communication capabilities from every participating node due to the fact that the agents should be able to execute on each node. Also, SensorWare intrinsically does not support heterogeneity in terms of integrating nodes that do not adhere to its hardware and operating system requirements. It also does not support communication with external applications.

2.2.2. Agilla

Agilla [17] implements mobile agents in WSNs by integrating them into a tuple-space model. Each node contains a tuple space and agents reside within it. Agents are programmed according to the application requirements and they are cloned to the tuple spaces of nodes from a base centre. Agents are cloned between tuples without the need for pre-configuration or pre-allocation of

memory. Also, a maximum of four agents can run on a single node. Every agent is programmed to perform a specific task. The tuple spaces of all nodes essentially implement a network wide shared memory.

Agents are adaptive to their environment at node level and can change their behaviours. Despite this, it features decentralized behaviours at a certain degree by for example preventing concurrent agent activity and managing collective data collection latencies. It does so by enforcing specific behaviours in the agents thus implementing decentralized application activities. Also, bottlenecks can be observed during deployment of multiple agents (cloning and migrating). Furthermore, agents in Agilla are simple and do not offer real reasoning capabilities. Also, the deployment is restricted to nodes running TinyOS [130] (heterogeneity). Finally, its low-level programming model creates hard to read and maintain code.

2.2.3. Impala

Impala [91] is designed for long running sensor networks. It uses an asynchronous, event-based middleware layer that compiles mobile agents into native code and deploys them in network nodes. The agents in native code are then linked dynamically in the node where they are deployed without interfering with applications that are already running. Impala provides application adaptation and new protocols can be applied at runtime. The agents in Impala enable the system to adapt to application requirements and network conditions by switching between protocols using an approach that implements finite state machines aiming to ensure scalability and openness.

The main drawback with Impala is that it does not support heterogeneity as it is designed to be applied on a single hardware platform (Hewlett Packard iPAQ Pocket PC running Linux).

2.2.4. Discussion

As we have seen above, the main drawback with middleware that use mobile agents as part of their functionality is that such programming approaches intrinsically support a single platform and they do not support heterogeneity. Other approaches of middleware with similar characteristics to the ones reviewed and enhanced by agents are described by Shakshuki et al. in [24], González-Valenzuela et al. in [87] and Freitas et al. in [76]. The task of programming decentralised nodes to exchange increased volumes of context data describing their environment as well as to coordinate their activities to perform sophisticated cooperative tasks is a complex one.

2.3. SAN Middleware Supporting the Integration with External Applications

In this section we will review SN middleware that enable interaction with an external to the SN system and potentially a MAS. Two types of integration are supported: task and data integration. When task integration is supported, the SN nodes and the external system are running applications and coordinating their tasks. In data integration applications the SN produces data to be used by the external system. The SOA paradigm (see 2.1.1) enables the implementation of such integrations through the representation of nodes in SN and external systems as services and service consumers. The “web services” are a messaging model that enables the deployment of a SOA and have been commonly used for WSN integration (see below). According to [57], they encapsulate existing and new applications in a way that the application communication will evolve from the paradigm of message exchanges to a new one that involves accessing, programming and integrating application services. The web service descriptions and messages use the XML format.

A SOA requires from services to provide with well defined interfaces for interaction and discoverability. The interfaces and the services are described

using the “Web Services Description Language” (WSDL) [132]. The WSDL typically describes the web services, while the “Simple Object Access Protocol” (SOAP) [125] describes the communications protocols. SOAP is used to implement the interaction between services. SOAP supports both synchronous OOM communication and asynchronous MOM communication. The XML schemas are used for marshalling types to and from programming language types making the web services language independent. We will now review a number of SN middleware that follow the SOA approach to implement integration with the potential of also integrating with external applications/systems (e.g. a MAS).

2.3.1. Open Geospatial Consortium (OGC) Sensor Web - NOSA

Sensor Web [35] was developed to support wired and wireless sensor networks. It is a paradigm for enabling heterogeneous sensors, actuators and devices to be discoverable, accessible and controlled via external applications and the web. It implements a service oriented middleware for enabling data collection and task management among heterogeneous sensors. It uses XML and SOAP to implement the protocols for the communication within the system. The Sensor Web Enablement (SWE) standard was created to support the OpenGIS Consortium (OGC). It includes a number of specifications such as two XML based languages: SensorML [123] for describing sensors and platforms and “Observation and Measurement” defining terms used for measurements. It also specifies three important services: the sensor collection service, the sensor planning service for managing sensors and the web notification service for asynchronous messaging using a variety of communication protocols to interact with web applications.

NOSA (NICTA - Open Sensor Web Architecture) [95] is a service oriented sensor web. The goal of this work is to support data access continuity over prolonged periods of time and for potentially large loads of data. It also

provides access and control of sensors that are exposed as services. It is open source architecture and it features service oriented architecture characteristics such as: abstraction concealing heterogeneous sensor platforms, interoperability and support for integration with external systems. This way the processing of the data can be shifted from the sensor networks to external applications with more computation capabilities. Besides the core SWE services NOSA includes a directory, a sensor coordination and data processing service [95], [35]. Currently it only supports nodes running the TinyOS operating system [130].

2.3.2. OASiS

OASiS [62] provides with a service oriented WSN development framework using a multilayer development process. The functions of the WSN can be viewed as a set of services that are offered by the middleware such as: service discovery, composition, QoS (Quality of Service) and failure detection, management of the networked nodes. In particular it features dynamic service discovery and supports heterogeneous platforms. Following the SOA paradigm, it can be integrated with external to the WSN systems and participate in the design of real world applications. We need to note here that it does not support actuators, which is an important feature of our requirements.

2.3.3. TinySOA

TinySOA [10] allows the development of WSN applications. The high-level of abstraction of TinySOA enables external applications to access nodes in a SN via a simple API that supports a variety of programming languages. This abstraction hides the complexity of the underlying WSN hardware and the details of the WSN implementation from developers of applications that require access of sensor data. Therefore it is suitable for easy integration with web applications enabling them to acquire information from the sensors. To simplify the integration task further, TinySOA offers a node discovery service typically running on a gateway component that can be seen as the bridge between the

low-level WSN and the external applications. TinySOA is limited to nodes that run the TinyOS operating system [130].

2.3.4. USEME

USEME [23] is a service oriented middleware for WSN (Wireless Sensor and Actuator Networks) applications. It features five components for service discovery, invocation and communication, real time constraint management, configuration and group management.

The discovery component includes a directory of available services allowing them to be discovered. The invocation and communication component translates external application commands into suitable communication messages between nodes. The constraint management sets priorities, maximum execution times. Finally, the middleware aggregates nodes sharing common functionalities into groups and manages the services of the created groups (group management component).

2.3.5. SIXTH

The SIXTH middleware [34] facilitates cross-platform and cross-service interoperability. It is JAVA based and implements a Sensor Web middleware. External applications (e.g. a MAS) would need to use adaptors to connect with SIXTH. The SIXTH API enables the creation of such adaptors. It implements the Open Service Gateway Initiative framework (OSGi) [118] that enables it to accept new connections dynamically at runtime. In order to support heterogeneity, SIXTH requires the creation of new adaptors for new sensors. Adaptors have been developed for TinyOS nodes [130], Arduino [102] and web-based data sources (considered as sensors in SIXTH) such as Twitter. Also, it supports dynamic sensor re-programming e.g. request a sensing for light when no motion is detected. A main requirement of the thesis is to create a systematic way for linking agent functionality with sensors and actuators while SIXTH focuses primarily on sensors.

2.3.6. Discussion

SOA provides with a way to connect heterogeneous WSN nodes (some supporting only sensors, others also actuators and devices) to external applications such as a MAS. Other middleware in this category are StreamWare [56] and MidCASE [97] and [94]. The middleware described in this section can generally provide the desired integrations. However, further application development would be required for more sophisticated applications that deal with the dynamic creation, management and monitoring of avatar relationships between software agents and hardware sensors/actuators at runtime. A separate, higher level middleware could systematize how this integration would be achieved in a transparent way for the system developers. The middleware reviewed in this section are still dependent on the underlying SN and they tend to focus on gathering information from sensors while the higher-level middleware would focus on using this information efficiently.

2.4. Pervasive and Robotics Middleware

We review next pervasive and robotics middleware that were selected according to the criteria of 2.1.2 and with the aim to achieve integration of agent AI to sensor and actuator networks. These types of middleware are looking at the integration from an application perspective that uses data and information stemming from SN as opposed to looking at it from the low-level data gathering perspective. Pervasive/AmI middleware can be used together with the SN/WSN middleware to implement efficient pervasive applications ([66], [29]). We also include types of middleware that represent different technologies and programming paradigms.

2.4.1. SALSA

SALSA - The "Simple Agent Library for Smart Ambients" or SALSA (Favela et al. [43] and Rodriguez et al. [65], [59]) implements the agent model

programming paradigm. It features a middleware and an architecture enabling the creation of autonomous agents that are reactive to the context of ambient computing environments. SALSAs was designed based on requirements from the healthcare domain [59]. It can be categorized as a MOM for the development of AmI applications that uses agents to enable interoperability between different hardware, software, devices and humans. Agents are used as proxies to communicate and exchange information. SALSAs uses an XML based language to describe services and protocols for enabling the communication between the agent-proxies. It also features an agent broker for managing communication amongst them. In particular it uses the Extensible Messaging and Presence Protocol (XMPP) [74] which is an XML based communications protocol for MOM and the “Jabber” open-source instant messaging server [65]. According to [65] the agent broker is capable of scaling to a high volume of streaming XML connections.

With regards to the problem we are trying to solve, SALSAs exploits MAS capabilities for creating a middleware for AmI. On the other hand, our goal is to extend MAS AI behaviours in the physical world for AmI and also to enable AmI to benefit from MAS computation and communication capabilities. Furthermore, we aspire to take away the complexity from the task of integrating software agents with physical avatars from the MAS, allowing the latter to apply its functionality to the physical world in a transparent way while SALSAs uses the software agents to enable the integrations.

2.4.2. Amigo

The European IST Amigo project [105] pursued the AmI vision by aiming to develop a networked smart home. A major requirement of the system was to ensure the integration of heterogeneous devices and to achieve interoperability. The heterogeneous devices include personal computing, mobile computing, consumer electronics and home automation devices. The Amigo middleware supports a number of SOA protocols for service discovery (such as UPnP [131] and WS-Discovery [116]) and interaction (such as SOAP [125]) ensuring

interoperability and presenting system designers with a choice from a selection of protocols. It allows the development of software as services that are delivered and consumed on demand [105]. It enables the loose coupling of autonomous services, allowing them to communicate within a dynamic smart home environment. The interoperability between the services that describe heterogeneous devices is made possible via a semantic based framework (underlying the SOA implementation).

Amigo consists of two main layers. The “Base Middleware” layer [105] deals with the networking and the communications. It implements a SOA and features discovery protocols and security mechanisms for authentication, authorisation and encryption. The “Intelligent User Services” layer acts as an integration broker between service providers and consumers. It also abstracts information from a variety of sources such as physical sensors, user activities and applications into high-level "contextual information". It will then use this high-level information to provide context aware services.

Amigo is used for the integration of heterogeneous devices from the domains of personal and mobile computing, consumer electronics and home automation. There is a subtle difference between Amigo and the aims of the thesis. We aim to systematize the integration of MAS agents with AmI architectures while Amigo is designed to achieve interoperability between hardware devices in an AmI context. Besides interoperability we investigate how to make devices appear smarter in the environment without further increasing their cost.

2.4.3. RoboCare

The RoboCare Domestic Environment [81] is an experimentation environment for AmI applications that mostly revolve around transparent monitoring and supervision. RoboCare implements a web service oriented architecture with software agents and physical hardware being regarded as services or sets of services. The software and hardware are integrated using a service oriented middleware that uses an event manager agent to route requests amongst the

services that represent software and hardware agents. There is a tracking and a monitoring component that along with the event manager they provide the Active Supervision Framework (ASF). They use a service oriented approach for low-level messaging but the mediation of the messages is performed at MAS level.

The middleware in RoboCare uses a service oriented approach for low level messaging but the mediation of the messages is performed at MAS level. In our case we want to hide the complexity of the mediation functionality from the agents and the hardware. In this way we can propose a middleware that can be used for creating applications using a variety of MAS systems.

2.4.4. iCore

An interesting approach was followed by the iCore project ([108], [78] and [75]) that develops a platform focusing on aspects of IoT infrastructures that deal with device heterogeneity, unreliability, resilience and the complexity associated with the huge quantities of usable objects in a smart city context. The architecture of iCore uses the concept of the virtual object (VO) to represent any real world object (RWO) similarly to the way software agent interactors represent physical world ones in the proposed in this thesis approach. The iCore platform uses “Restful” web services [55] to enable the VOs to interact with RWOs and the functionality of VOs is aggregated to create composite VOs (CVOs) that are the equivalent of the software agents in our approach. In general a CVO that interacts with RWOs is not dissimilar in concept to the UA of the thesis.

The iCore project produced a platform that implements an IoT infrastructure. In the thesis we aspire to provide with a systematic way toward the creation of middleware that can be used for creating platforms like iCore. One key difference with iCore is that we want to enable system developers to use a MAS of their choice as long as it fulfils (subject to certain requirements as we will see in the following). The iCore platform uses “Restful services” [55] to allow the

VOs to connect to and control the physical objects. The approach is tailored to the VO layer of iCore that consists of the specific types of iCore VOs. The challenge that we confront deals with linking agent sensors and actuators from a variety of MAS to physical objects combining them into complex cognitive and communicative entities (UAs).

2.4.5. PEIS

The PEIS (Physically Embedded Intelligent Systems) ecology [6], [7] provides a framework for incorporating smart devices into AmI systems. The PEIS-Ecology approach is applied in the areas of domestic and service robotics focusing on the creation of a network of cooperating robotic devices as opposed to using a single powerful and competent robot [7]. According to [7] the cooperating robotic devices (PEIS) are distributed in the smart home environment and they can be: sensors, actuators, smart appliances, RFID-tagged objects or mobile robots.

The approach utilizes the PEIS middleware by which devices of different types and capabilities (including mobile robots) can cooperate. The PEIS approach implements a distributed tuple space that is deployed upon a peer-to-peer network and follows the publish-subscribe paradigm. Different PEIS publish information as tuples. PEIS clients can join and leave the system dynamically. The approach focuses on simplifying the communication between the devices for smart home applications that use data from a variety of sensors in a tagged environment. Our proposed approach presents integration potential to such a system by providing it with a link to intelligent agent behaviour.

2.4.6. The “Middle Layer for Incorporations”

The “middle layer for incorporations among ubiquitous robots” by Kim, Choi and Lim [92] connects physical and simulated sensor-actuator behaviours. The middle layer consists of a sensor and a behaviour mapper. The first helps simulated robot-agents obtain physical sensor information from mobile robots

while the second allows software robots to present physical behaviour. The “middle layer” does not focus only on providing software agents with access to the physical world, but it is also capable of the opposite, i.e. providing mobile robots with access to virtual resources. A shortcoming of the middle layer is the fact that as soon as the mapping table is created from the sensor/actuator mapper, it is not updated anymore. This static setup would create problems in dynamic environments such as in AmI systems where sensors, actuators and devices connect and disconnect at runtime. Furthermore, the “middle layer” works only with mobile robots.

2.4.7. MARIE

MARIE (for Mobile and Autonomous Robotics Integration Environment) [14] is a middleware platform that enables interoperability and distributed control of sensors and actuators that are generally applied in the context of robotic applications. It enables the integration of new and existing software for rapid prototyping of robotic applications in a distributed environment. MARIE follows the mediator design pattern providing mediator interoperability layers among applications. Thus, the key features of MARIE are the interoperability and reusability of robotic application components as well as the independent interaction with each connected application.

MARIE uses the ACE library [129] for the transport layer (TCP/IP), operating system functions such as threads and processes, operating system interoperability and real-time support [14]. The main idea is to reuse code from existing robotic applications to create an integrated system (as long as MARIE knows how to interact with the particular code). The MARIE middleware uses four functional components for interaction and communication between the applications through a centralized control unit (implemented as a virtual space) that implements the mediator functionality. The four components are: the application adapter, the communication adapter, the communication manager and the application manager. The application adapter is responsible for the communication between the control unit and the applications. Its purpose is to

enable applications to integrate with the system. Furthermore, the communication adapter translates information allowing different applications to exchange data correctly [14]. The communication manager manages the communication between the different applications and the application manager is responsible for managing the whole system.

MARIE does not offer automatic configuration and it features a static communication setup. Therefore it is not capable of coping well with the requirements of dynamic scenarios such as in AmI/IoT applications and it would not be suitable for our application.

2.4.8. The Player/Stage project

The Player/Stage project [12] features the simulation of multi-robot behaviours in virtual environments and the ability to reproduce the mechanical behaviours of the robots. It consists of two components the “Player” and the “Stage”. The Player component uses a repository of interfaces and drivers. A driver in this context refers to a software program for operating a sensor, actuator or robot that is supported by the middleware. The interface allows a client software application in a virtual environment to control a physical actuator or receive data from a sensor using the appropriate driver.

The Stage component is a graphical simulator that is used for modelling devices. The clients are software programs using the Player component that acts as a middleware linking them to physical robots, sensors and actuators. The clients are developed using language specific libraries for different programming languages (such as C, C++, Java, and Python) enabling access to the Player. The Player serves as an interface to many different types of robotic devices and provides drivers for many hardware modules. The Player does not operate on an abstract level like many of the middleware we have seen so far (e.g. SOA based middleware) and integration tasks with new hardware and software present significantly higher complexity. Updating the Players’ library requires the creation of new drivers/software for the new hardware.

2.4.9. ReMMoc

ReMMoC (Reflective Middleware for Mobile Computing) [73], is a web-services based reflective middleware. It was designed with the intention to adapt discovery and interaction protocols dynamically as per the requirements of the current mobile service environment aiming to overcome platform heterogeneity. It uses reflection, to select dynamically the most appropriate communication protocol according to the context [73].

It provides a dynamically reconfigurable binding mechanism that allows clients to bind and interoperate with services that are implemented using a variety of communication models such as RMI, CORBA and “publish-subscribe”. If for example a new CORBA service is discovered, the binding framework will dynamically reconfigure itself to act as a CORBA client.

ReMMoC uses the web services (WSDL documents [132]) abstractions. This way the service consumers (clients) can interact with services (providers). This abstraction is then mapped onto the appropriate protocol at run-time and services are invoked remotely by their clients.

ReMMoC was designed to reside on mobile devices and it implements an API for performing service discovery and service interaction that is independent of protocol implementation (SOAP [125], XML-rpc [19] etc.). According to the ReMMoC middleware, interoperability in open wireless networks should be managed by the networked devices themselves. The main constraint is that service consumers (client applications) should be developed using the ReMMoC middleware. ReMMoC is only focusing on clients. This means that different ReMMoC clients can interact with a service that is using a different communications protocol but non ReMMoC clients cannot interact with services that are based on a different communication protocol (the services do not have an interoperability layer in ReMMoC).

2.4.10. Dynamic-TAO & CARISMA

Another example of a middleware using reflective functionality is the dynamic-TAO [28] project. It uses reflection techniques to reconfigure Object Request Broker (ORB) components [86] at run-time. The CARISMA [54] middleware provides developers with an abstract syntax framework to profile applications. Interoperability is achieved when applications use their profiles to interact with each other. It implements reflection for enabling applications to change their profiles during runtime. What follows is a more detailed overview of the ReMMoC middleware in order to provide with a better view of the reflective middleware paradigm.

Reflective middleware such as ReMMoC ([72], [73]), dynamic-TAO [28] and CARISMA [54], were initially developed to enable the OOM to overcome their shortcomings especially with regards to interoperability amongst heterogeneous devices that support different communication protocols and with regards to implementing loose coupling. Also they were used for self-monitoring. Reflection had shown great potential but currently the development of MOM, the SOA paradigm and the ESB technologies are more commonly used for implementing the above behaviours (namely interoperability and loose coupling). Other reflective middleware and in particular in the context of SN/WSN are Milan [94] and the Impala middleware [91] that was described in 2.2.3.

2.4.11. Discussion

In this section we reviewed a number of middleware that could potentially be used for integrating MAS directly with sensors/actuators or via cooperation with WSN and SN low-level middleware. This category of middleware presents potential for at least partially satisfying the aims of this thesis as we will see in the next section.

2.5. Evaluation

Table 2-1 shows the key technologies used by each of the reviewed middleware, along with useful middleware features for the task of linking agent functionality to sets of physical sensor and actuators. We will use this table as a point of reference to evaluate which features are relevant to the thesis and also to determine whether existing work can achieve the aims of the thesis.

Middleware	Domain	Programming Paradigm	Interface	Heterogeneity	Reflection	Built-in intelligence	Inter-process Communication	Transparency
SensorWare	WASN	database, agents	declarative	no	no	basic	asynchronous, tuples	low-level
Agilla	sensor & actor Networks	agents	coding (imperative)	no	yes	basic	asynchronous, tuples	low-level
Impala	WSN	agents, event driven	-	no	yes	basic	asynchronous	no
SIXTH	WSN	component based - OSGi	declarative	yes	no	no	asynchronous	high-level
Sensor Web - NOSA	SN, WSN	SOA	declarative	no	no	no	asynchronous-OGC - broker- OOM	high-level
OASiS	SN, pervasive	SOA	declarative	yes	no	no	asynchronous	high-level
TinySOA	WSN	SOA	declarative	no	no	no	base station (broker - asynchronous)	high-level
USEME	sensor & actor Networks	SOA	declarative	no	no	no	synchronous & asynchronous	high-level
SALSA	Aml	agents, SOA	coding	yes	yes (adaptive)	basic	asynchronous, MOM-broker	high-level
Amigo	pervasive, Aml	SOA	declarative	yes	no	no	asynchronous, MOM-broker	high-level
RoboCare	pervasive, Aml	SOA, agents	declarative	yes	no	no	asynchronous, MOM-brokers	high-level
iCore	IoT	agents, SOA	declarative	yes	no	no	asynchronous, MOM-broker	high-level
ReMMoc	pervasive	pub-sub, web services	declarative	yes	yes	no	asynchronous, MOM - broker	high-level
PEIS	pervasive	pub-sub	coding	yes	no	no	tuples	high-level
Middle Layer	robotics, Pervasive	pub-sub	declarative	no	no	no	asynchronous, MOM - broker	high-level
MARIE	robotics, pervasive	mediator	declarative	Yes	no	no	asynchronous, broker	high-level
Player	pervasive	message queues	coding	Yes	no	no	asynchronous, MOM - broker	low-level

Table 2-1. Cross-reference of approaches and middleware characteristics of middleware that support AI by using agents as part of their implementation or via integration.

We have reviewed a number of representative middleware (see criteria in 2.1.2) that purport to support intelligence by using agents as part of their implementation. With regards to the aims that were set in the introduction, agent-based middleware is usually tied to a single agent platform and therefore they cannot support multiple ones without integration with external applications.

Furthermore, recall that one of the aims of this work is to make existing devices in a sensor and actuator network appear smarter. In situations where connectivity is limited, distributing the reasoning or localizing the reasoning on every node (e.g. in WSN where nodes are not always within the reach of servers) would pose a feasible solution. Implementing sophisticated intelligent applications is a challenging task due to hardware limitations or due to the complexity of programming distributed simple agents to coordinate their actions to support sophisticated and intelligent tasks. In domains such as smart homes, healthcare systems, museums there are commonly facilities for centralized processing and the sensors and actuators are usually either static (wired) or wireless within environments with strong and reliable connectivity. In such situations distributed or node processing is not as vital and we could use simpler sensors and actuators in terms of processing and memory capabilities with a potential impact on the cost. Also, another general observation is that agent based middleware approaches are usually implemented with an intrinsic support of a single platform overlooking heterogeneity issues, for example see [8], [91] and [17].

After reviewing middleware that incorporate intelligence in the form of agents, we proceeded with middleware that allow the connection of the low-level WSN/SN/SAN to external applications. A relevant external application in this context would be a MAS. Middleware such as the reviewed [95], [62], [10], [23] and [34] have a proven record of integrating WSN/SN/WSAN with external applications. In particular, we saw that it is common practise to use SOA for integration and interoperability among heterogeneous devices. Therefore such middleware would be able to provide the necessary connectivity between software agents and SAN.

We also saw that the WSN middleware focus on connectivity and interoperability issues as well as on the low-level tasks of gathering information from sensors and the controlling of actuators. In this thesis we aim to use the data gathered by sensor/actuator networks to support a specific agent tasks.

These tasks will enable software agent bodies to sense and act in the physical world using avatars construed as aggregations of sensors and actuators in sensor/actuator networks. The aforementioned middleware can achieve this but not in a systematic or a transparent way.

In this work we aim to systematize this process and make the creation of such applications transparent to the system developers. In order to achieve this we would require a middleware that looks at the integration from an application perspective in order to systematize how this integration would be achieved. At the same time we want to abstract from the integrations' functionality in order to make it transparent to the developers.

We also reviewed pervasive/AmI middleware that specialize in similar applications in order to evaluate whether there is existing work that achieves this and also to build upon existing research to propose our own approach. Note that the list of middleware presented is not exhaustive but we carefully selected them based on the criteria of 2.1.2 and based on their potential to achieve the integration of agent AI to sensor and actuator networks. Table 2-2 evaluates each reviewed middleware against the aims of the thesis.

Middleware	Aim 1 MAS-sensor/actuator/SAN integration capability	Aim 2 systematic integration of MAS with sensors/actuators/SAN	Aim 3 transparent integration of MAS with sensors/actuators/SAN	Aim 4 multiple MAS support	Aim 5 heterogeneity
SALSA	yes	yes	no	no	yes
Amigo	yes	no	yes	yes	yes
RoboCare	yes	no	yes	no	yes
iCore	yes	yes	no	no	yes
ReMMoc	no	no	no	no	yes
PEIS	no	no	no	no	yes
Middle Layer	yes	no	yes	no	no
MARIE	no	no	yes	yes	yes
Player	no	no	no	no	yes

Table 2-2. Evaluation of pervasive/AmI middleware in relation to the aims of the thesis.

The reviewed middleware implement pervasive applications and similarly to the previous set of reviewed middleware (SOA integration SN middleware) they could potentially achieve connectivity. We have identified positive aspects of the middleware towards our intended use and also recorded concerns, usually stemming from the fact that they were not designed for the specific application of the thesis.

Other pervasive middleware that support integrations but do not fulfil all of the aims of the thesis at the same time include the ones described in [47], [15], [63]. We conclude that there is paucity of frameworks defining a systematic way for linking the computation and functionality of intelligent agents to networked sensor/actuation devices in a transparent way enabling them to manifest their behaviours in the physical environment while at the same time fulfilling all of the aims that were set in the Introduction. Therefore, we find an opportunity to build upon experience gained from the reviewed research in this chapter and propose our own approach that will start the discussion for creating a middleware that satisfactory fulfils all the aforementioned aims.

2.6. Summary

In this chapter we reviewed a number of middleware and approaches for adding intelligence to sensor and actuator networks. We have also identified a paucity of frameworks defining a systematic way for achieving all the aims of the thesis due to the fact that the reviewed approaches were not designed for the specific application of the thesis. The following chapter will present a middleware architecture that capitalises upon the current discussion regarding the reviewed middleware and their corresponding implementations.

3. REQUIREMENTS & ARCHITECTURE

We have identified a paucity of middleware approaches that simultaneously satisfy all the aims of the thesis and in this chapter we will define a set of requirements and propose an architecture for a middleware that aims to fill this gap. We will craft our solution based on the reviews of the current state of the art and the evaluations of the relevant technologies as presented in the previous chapter. This chapter starts with this evaluation leading to an overview of the proposed approach. A description of the physical and software agent sensors and actuators of the approach follows. We then proceed with defining the requirements of the proposed solution that are used for the design of the proposed middleware architecture. We then identify and justify the use of a formal framework for the specification of the middleware in the next chapter.

3.1. Overview

Architecturally the proposed system consists of four layers, the:

- Physical world environment layer: it can be any physical environment that contains physical sensors/actuators or sensor/actuator networks as implemented by low level middleware that also enable the sensors and actuators to be exposed to external to the SAN applications (see middleware in section 2.3). The external application in this case would be the middleware of the thesis.
- MAS layer.
- Middleware layer that allows the sensors, actuators and agents to interact with it via an API in order to implement UAs.

- Application layer: In our case we focus on smart home applications and in particular on enabling agents to add AI to SAN within smart homes.

Figure 3-1 provides an overview of the system.

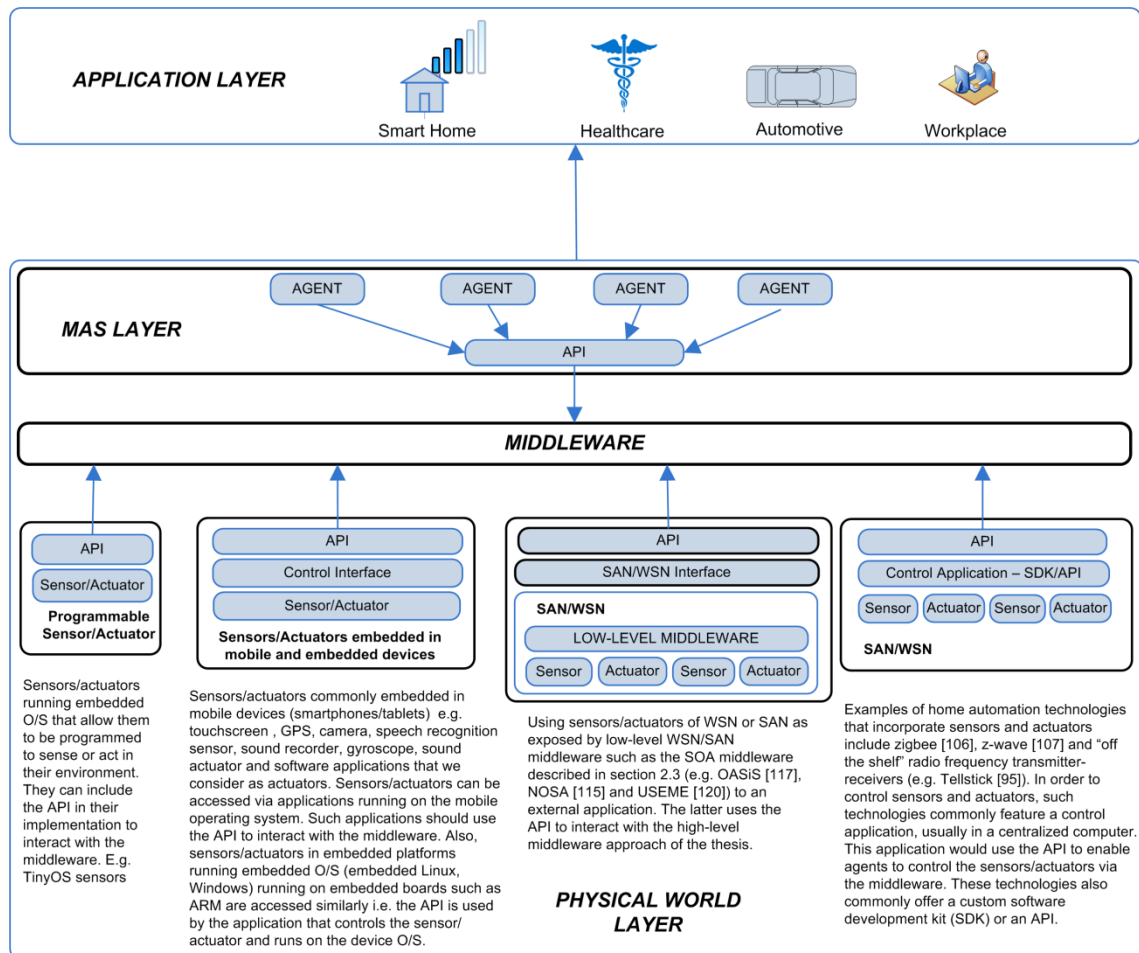


Figure 3-1. We see an overview of the proposed system architecture. In our approach, any external data source that uses the API is considered to be a sensor. Similarly, any external to the middleware resource that can produce an effect in the physical world and uses the API is considered to be an actuator. Accordingly, our approach considers agent sensors as code that internalizes sensory data to the agent mind/reasoning component from the API and actuators as code that enables this component to interact with the API in order to create control messages (section 1.4). The technologies referenced in the above figure are exemplifying what could potentially use the API to interact with the middleware. In 3.2.2 we describe how these technologies use the API.

Our proposed approach aims to satisfy the aims of the thesis. As we saw in the background section, according to literature sources that include Papazoglou ([58], [57]), Erl [89] and the OASIS [115], interoperability is an intrinsic characteristic of architectures that follow the SOA principles that were described in 2.1.1. Therefore the SOA paradigm has been used extensively to create middleware approaches that allow heterogeneity such as SALSA [59], Amigo [105] and the middleware frameworks of section 2.3. A SOA therefore could allow the integration of heterogeneous devices and heterogeneous applications such as multi-agent systems. We will use the SOA paradigm to implement a middleware framework enabling the integration of agent and sensors/actuator functionality in order to satisfy the aim for multiple MAS platform and the aim for heterogeneous devices support.

Furthermore, the SOA principle for abstraction and in particular for concealing the implementation and the details of the provided by the services capabilities from the consumers of the services and the distributed system environment is a step towards transparency.

We want to further abstract functionality and reason about the system. In particular we want end developers to think in terms of agent bodies and avatars when designing their systems instead of services and low level software and hardware. Then they should be able to interact with the middleware in a transparent way using a declarative service oriented interface in order to implement their applications.

The “transparency” refers to the requirement that the application developer should be able to describe the agent and avatar bodies and register them to the system using a common method e.g. as services and then the system should take care of discovery, binding between the aggregations of services and the agents and communication management between the agent and the avatar bodies without the application developer being exposed to how this is done. This entails a way for the middleware to abstract aggregations of services in a way that they can be discovered by agent bodies that they will use them as

avatars. It also entails the management of these agent to avatar relationships that are implemented using message exchanges.

A reflection layer in the middleware could implement the abstract aggregations of services and agent bodies within the middleware by creating a reflection of the overall system based on context from SOA services and reason on this model in order to implement and manage UA relationships. In 2.1.1 we saw that there are two levels in a reflective application, the base-level and an internal representation/model of the base-level that is called the reflection-layer. In our approach, the base level would describe the service providers and consumers that correspond to the avatar and the agent interactors. Furthermore, we saw that in the reflection approach the entities of the reflection-layer model the base-level objects (Figure 2-4). This way we can create internally models of the agent and avatar sensors and actuators.

According to [33] and [27] the base-level deals with the usual functionality of the application within its problem domain which in our case includes describing agent and avatar sensor/actuator functionality as service consumers and service providers respectively, implementing their communication protocols with the middleware and enabling them to register with it. The reflection-layer on the other hand reasons about the base-layer in order to be able to enrich, monitor and potentially change the behaviour of the application during runtime (Figure 2-4). In our case the reflection layer would process the models of the interactors, use them to create models of agents and avatars and implement transparently more complex discovery functionality for binding agent bodies to avatars. Changes in the base layer e.g. a message from an agent interactor will be reflected in changes in the reflection layer and the latter would this way manage the message exchanges between agents and avatars.

Our approach should use SOA and reflection to implement a middleware that will enable the integration of agent functionality with SAN this way satisfying the first aim of the thesis (see 1.3). To satisfy the second aim of the thesis, the middleware should integrate the agent functionality with avatar bodies in a

systematic way. We will use the SOA paradigm to implement a systematic way for enabling agents and sensors/actuators to connect to the middleware using an API that should entail:

- Describing each interactor as a service provider (if it is a physical sensors/actuator) or a service consumer (if it belongs to an agent) using metadata.
- Using API functions to initialize a connection with the middleware.
- Using the API to interact with other interactors via the middleware.

System developers should be able to use the API as described above in order to integrate agents with avatars (create UAs) without having to deal with the low-level details of how these integrations are achieved. The middleware internally should deal with the following:

- Implement service registration. Agent interactors should register as service consumers and avatar interactors as service providers.
- Perform discovery of compatible agent and avatar bodies dynamically. We remind that in our approach an avatar/agent body is an aggregation of sensors and actuators as per the definition of 1.4. Therefore the discovery involves the matching of a set of service consumers in an agent body to a set of service providers in an avatar body.
- Bind interactors of agent and avatar bodies; specifically bind each service consumer in an agent body to a service provider in an avatar body.
- Managing the message exchange between the bound to each-other interactors.

The above middleware functionality should be transparent to the system developers that use the middleware (thus we satisfy this way the transparency aim of the thesis). The proposed approach is striving for a lightweight solution with regards to the resources it requires from the participants of the system aiming for the minimum amount of added complexity and resource

consumption from them. The goal is to build our framework based on mature and commonly used middleware technologies such as SOA making our solution more portable thus enabling engineers that attempt to do something similar to reproduce and benefit from it.

A description of a simple scenario would be useful in terms of introducing the basic concepts of our proposed architecture and the way we would use it to create an application. Consider an end user aspiring to empower his networked (in a LAN-Local Area Network) devices with MAS intelligence. For simplicity we assume that the user's network consists of a single remote control light switch and a single wireless door sensor that senses whether a door is opened or closed. They are both controlled remotely by software running on a computer using radio frequency communications. For this example let us consider the control software, radio frequency communication mechanism and proposed remote controlled light switch and door sensor that are described in [127]. The first step of the end user towards realizing the system would be to call a consultant whom we will call the "system designer".

The proposed architecture considers the light switch and the software (see [127]) that controls it as an actuator (the same applies for the sensor). The system designer will create metadata descriptions for the sensor and the actuator. The end user should also inform the system designer of what he would like the system do. For our simple example let us consider a simple task for the system such as changing the state of the light (switching it on or off) every time the door opens and closes.

As we will see in the following we consider for our architecture agents that feature a decision making component (see [44]) and a set of agent sensors and actuators for interacting with the agent environment. For the purposes of the described scenario we assume using the GOLEM platform [82] to create such agents. The system designer would use the GOLEM platform in order to program an agent bearing a decision making component, an agent "door" sensor and an agent "light" switch actuator. When the decision making component receives a sensing event via its agent "door" sensor it will issue a command to

switch the light on or off using its agent actuator. The designer will then create metadata describing the agent sensor that senses the state of a door through receiving sensory data in the form of the values "OPEN" and "CLOSED". Similarly, the designer will create metadata describing an agent actuator that sends command messages for switching a light on or off.

The agent, the light switch (with control software) and the door sensor (with control software) should be connected to the same local area network. In this setting the system designer is now ready to apply the middleware based solution that is proposed in the thesis. This entails connecting the middleware to the same network and enabling the agent and physical sensors and actuators to connect to the middleware using an Application Programming Interface (API) for sending and receiving messages (we will describe the API in the following sections).

The end user can now start the system and the following should happen:

- The agent and physical sensor/actuator metadata descriptions will be sent to the middleware via the API.
- The middleware will process these descriptions in order to determine which connections to make between the agent and physical sensors and actuators. In other words it will process the metadata descriptions to connect (bind) the agent "door" sensor to the physical door sensor and the agent "light" switch actuator to the physical one forming a Ubiquitous Agent.
- The middleware will enable and manage the communications and interactions between the connected agent and physical sensors and actuators.
- The agent will apply its policy. Every time the door opens/closes the door sensor will be producing a message that will be received by the agent "door" sensor of the agent via the middleware. The virtual door sensor of the GOLEM agent will notify the decision making component which in its turn will instruct the agent actuator to send a switch command. The command will reach the physical light switch via the middleware and the switch will turn the light on or off accordingly.

The proposed architecture should cope with replacing sensors and actuators with ones that present the same functionality without requiring re-configurations or system changes. In the proposed approach, adding new or different types of sensors/actuators would require new metadata descriptions and possibly programming the agent with new decision policies implementing new scenarios. We can scale the above example by adding more physical sensors, actuators and more agents that communicate with each other in order to implement complex scenarios. In the following we describe the requirements for a MAS and a physical world environment that contains connected sensors and actuators in order to participate in the proposed architecture.

3.2. MAS and Physical World

In this section we describe what we assume for the MAS and the physical world interactors in order to participate in the proposed approach.

3.2.1. MAS Scope

We are particularly concerned with software agents that separate intelligence via goal directed reasoning and decision making from capability allowing them to sense and act in a MAS electronic environment. The intention is to extend this capability for interacting with the physical world. A requirement for the MAS is that the agents should be residing in a MAS environment that is rich in computational resources and communication capabilities. MAS platforms vary in terms of the implementation of perception and action execution. Indeed, agent platforms like GOLEM [82] and CaRTAGo [5] use agent sensors and actuators explicitly as components of an agent's body, while platforms like JADE [25], [26] use them implicitly as part of sending and receiving messages.

With regards to the agent architectures considered by our approach, we assume that there will be a straightforward correspondence between agent perception/action execution (which may be manifested as a message exchange) and the sensing/acting of avatars.

Our approach has been inspired and motivated by the agent platform GOLEM [82], in particular, to enable agents deployed in the platform to access the physical environment. GOLEM is based on ideas of the PROSOCS [50] platform to support the deployment of three main entities: agents, objects and containers. Agents are cognitive entities that can interact with each other and with objects in the agent environment. As shown in Figure 3-2, agents are composed of two main components: (a) a declarative mind that contains the decision making of the agent and (b) an imperative body that contains the sensors and actuators of the agent.

The mind bestows, in addition to decision making, other reasoning capabilities to the agent such as planning and temporal reasoning [82]. The mind is situated in the agent environment within the agent body component. This component contains sensors enabling the agent to sense the virtual environment and actuators (effectors) to act upon it. In other words, sensors and effectors represent the interface between the agent environment and the agent mind. The software component that links the mind with the sensors and effectors in the GOLEM agent architecture is called the “brain”. The agent interaction is mediated by the environment, which is an evolving composite structure supporting agent-agent and agent-object interaction.

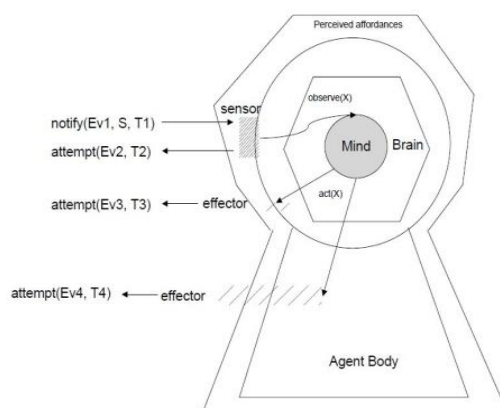


Figure 3-2. The architecture of an agent in GOLEM (borrowed from [82]). Interaction in the platform is event-based. Actions that happen in the MAS environment notify the body sensors. Similarly, actions of the agent via the body’s actuators generate events represented as attempts of action in the MAS environment. Such attempts happen if the action is possible.

Besides agents, our approach requires interfaces which in our architecture are defined as software allowing the interaction of MAS entities with external to the MAS code. Such interfaces in GOLEM can be treated as objects that can be interacted with by the agent body. They then transform this interaction into a second interaction with an external resource. Furthermore, the nature and the format of the data that the MAS recognizes and uses could be incompatible to the format of the data of the external resource. Therefore a mechanism is required that would also transform the data being exchanged with the external resource into the appropriate formats.

GOLEM objects are reactive entities. They have a trigger to receive events from the environment and an emitter to produce reactions to such events [82]. The trigger and the emitter ensure that the interaction between the object and the environment is completely asynchronous. An object is composed by an external object which is connected to the trigger and the emitter, and its purpose is to hide the complexity of an internal object, which could represent an external to the electronic environment of the agent resource. The general idea behind the internal object is that it wraps in it a resource of the external environment, thus hiding from the agents the complexity of interfacing with an external resource [82] (e.g. a web service).

Agents and objects are situated within containers. A container represents a portion of the agent environment and it works as a mediator for the interaction taking place between agents and objects. Events describe what happens in the agent environment as a result of actions being executed by effectors. When an event occurs, the agent environment notifies those sensors capable of perceiving it. The perceived events encapsulate data produced as the result of: a) "speech acts", which are actions by agents enabling them to communicate with each other, b) "sensing acts", which are essentially attempts by the agent to perceive the environment actively and c) "passive sensing", which can be described as the reaction of the environment to an event, by notifying instantaneously all the types of sensors that are capable of detecting it.

We described GOLEM as a platform that can be used in this thesis to provide a concrete example of a platform that features all the required functionality. GOLEM will be used in the case studies of Chapter 6.

3.2.2. Physical World Scope

Our approach considers sensors any data sources that present networked connectivity and are able to use the API, including software applications. Similarly any effector, software or device that uses the API and can produce an effect in the physical world is considered by our approach as an actuator. The API can be used by the sensors and actuators in a variety of ways (also see Figure 3-1):

- Directly by the sensor/actuator. In this category we have sensors/actuators running embedded O/S that allow them to be programmed to sense or act in their environment. They can include the API in their implementation to interact with the middleware. Examples include devices running the TinyOS operating system [130].
- Mobile devices running mobile O/S such as smartphones and tablets commonly embed sensors and actuators. These platforms (including Windows Mobile, Android and IOS) feature sensors and actuators such as a touchscreen sensor, a GPS sensor, a camera, a speech recognition sensor, voice/sound recorder, accelerometers, gyroscope, music/sound player actuators and software applications that can be regarded by our approach as actuators. For example an email/SMS sending application that can be controlled by an agent would be considered as an actuator. The above list is indicative of what we can do with such platforms and is by no means exhaustive. All such sensors and actuators can be accessed via applications running on the mobile operating system. Such applications should use the API to interact with the middleware enabling them to be accessed by agent functionality.

Sensors/actuators in embedded platforms consisting of embedded O/S (embedded Linux, Windows) running on embedded boards such as ARM [11] are accessed similarly i.e. the API is used by the application that controls the sensor/actuator and runs on the device O/S. An example of this and in particular of using the Raspberry Pi [126] platform with Arduino hardware to create and access sensors can be found in [16].

- Using sensors/actuators of WSN or SAN as exposed by low-level WSN/SAN middleware such as the SOA middleware described in section 2.3 (e.g. OASiS [62], NOSA [95] and USEME [23]) to an external application. The latter uses the API to interact with the high-level middleware approach of the thesis.
- Examples of home automation technologies that incorporate sensors and actuators include ZigBee [133], z-wave [134] and “off the shelf” radio frequency transmitter-receivers (e.g. Tellstick [127]). In order to control sensors and actuators, such technologies commonly feature a control application, usually in a centralized computer. This application would use the API to enable agents to control the sensors/actuators via the middleware. These technologies also commonly offer a custom software development kit (SDK) or an API.

The above provide with an indication of what could in principle use the API to interact with the proposed middleware. Finally, an “avatar” is a conceptual grouping of physical interactors that is controlled by a single agent.

3.3. The Middleware Requirements

We wish to develop a middleware component that takes agents from MAS environments such as the ones described in Section 3.2.1 and link their functionality to avatars that are built from sensors and actuators such as the ones described in section 3.2.2 in order to create UAs (see Figure 3-3).

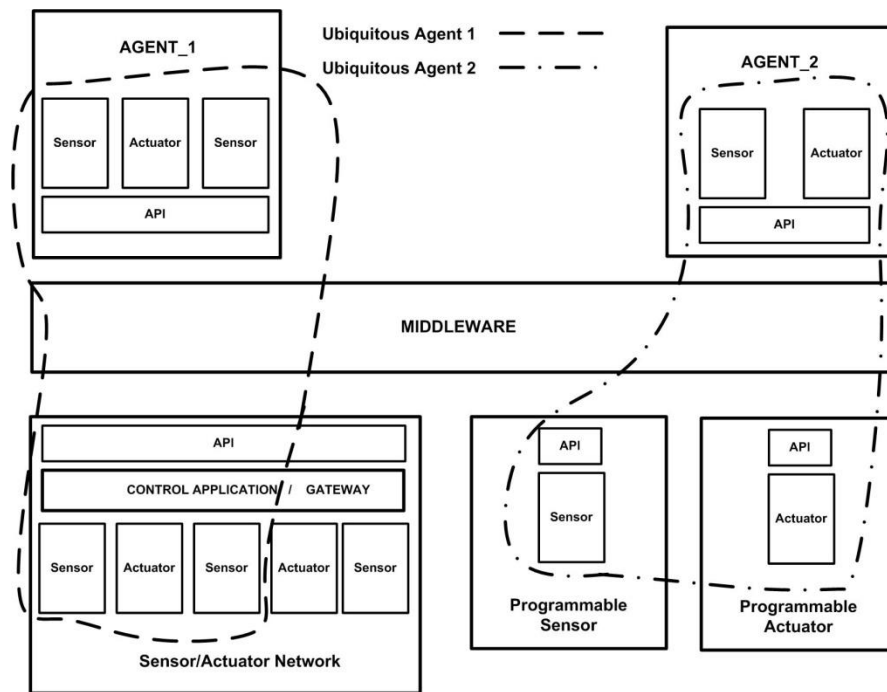


Figure 3-3. Middleware for Ubiquitous Agents.

The middleware will be described in terms of its desired functionality using requirement definitions and diagrams. The requirements will also define an API (Application Programming Interface) for enabling distributed physical and agent sensors and actuators to interact with the middleware. In this context, the API implements the endpoint for communication with the middleware. Agent and avatar sensors and actuators use the API to interact with the middleware independently. The avatar and agent interactor software that connect to the middleware may assume the role of the service provider and the role of the service consumer respectively.

3.3.1. The Life-Cycle of a UA

The life-cycle of each UA (Figure 3-4) in the middleware consists of the stages of “binding” and “UA session”. The life-cycle starts after the registration of agent and avatar bodies to the middleware. A body is a set of interactors that belong to a particular agent or avatar. The middleware implements discovery by finding compatible agent and avatar registered bodies and binds them together in order to start communication sessions between them. A communication

session is the quintessence of a UA. As soon as the UA session is stopped (termination) the bound instance of the UA is freed and the UA ceases to exist.

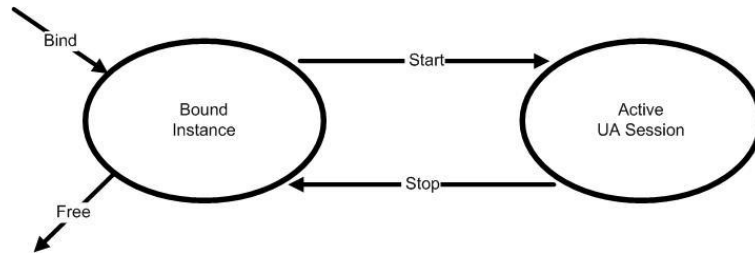


Figure 3-4. The life-cycle of a Ubiquitous Agent.

In the following we will see an in more depth description of what is required from the middleware to create a UA and what is required for each stage of the UA life-cycle.

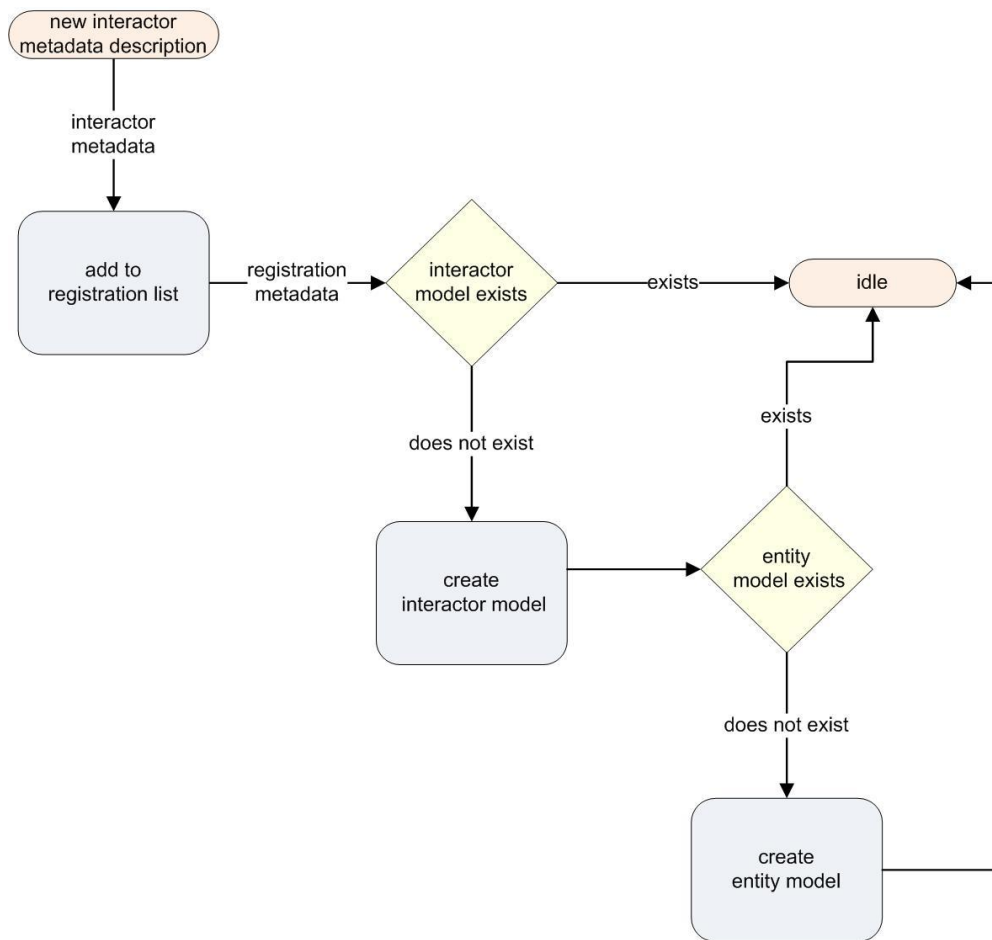
3.3.2. Registration

Agent or avatar body interactors connect to the middleware using the middleware API. The latter also enables the acquisition of a metadata description (e.g. from the hard disk) for the interactor as well as the messaging between the interactor and the middleware. The descriptions, the messages and the protocols used by the API should be implemented by using a meta-language. WSDL [132] is an example of such a language, despite the fact that we did not use it in the proposed implementation as we will see in the following. The system designer (see section 3.1) is responsible for creating the metadata descriptions and adding the necessary information in them. The following illustrate what the metadata should include:

- A unique identifier for the interactor.
- A unique identifier for the agent/avatar with which the interactor is associated.

- A list of all the unique identifiers for all of the sensors and actuators that constitute the body of the particular agent/avatar. By enumerating the unique identifiers of interactors in the metadata description, system designers are able to describe a set of interactors as an agent or an avatar body. Especially in the case of physical avatars, the system designers are able to describe any set of physical interactors as an avatar body.
- The “binding type” that instructs the middleware on how to bind agents with avatars. Binding refers to the establishment of connections between agent and avatar interactors within a UA. “Targeted” binding connects agent interactors to specific avatar interactors while “agnostic” binding allows the middleware to decide which agent will connect to which avatar interactors.
- If the binding type is “targeted”, the metadata should also provide with the necessary management information that will instruct the middleware which bindings are allowed to be performed. Otherwise the binding must be set to “agnostic”.
- The “service type” indicating whether it is a provider of a service or a consumer. For example the light switch of the scenario that was described in 3.1 is described as a service provider and the agent “light switch” actuator as a service consumer.
- A description section of the metadata that describes the functionality of the particular interactor in an abstract way as a service provider or consumer. The architecture assumes that agents and avatars share ontology so that e.g. “switch on” on one side means the same thing on the other.

Flowchart 1 shows the registration process for a new interactor like the door sensor of the scenario that was described in 3.1.



Flowchart 1. Registration of a new interactor to the middleware.

All metadata coming from the distributed interactor software via the middleware API is stored by the middleware in a data storage component that we will call the “registration list”. As we can see in the Flowchart 1, the middleware processes the registration list in order to use the stored metadata descriptions for the creation of internal representations of agents, avatars and their interactors which we will call models. The interactor models (stored in a data structure called the “interactor model repository”) consist of two layers: an information layer and an interactivity layer. The information layer stores the unique identifier of the interactor, a description of the interactor based on the metadata that is stored in the registry and also the type of binding that the interactor supports. The interactivity layer enables this model to exchange messages with other interactor models within the middleware. It should also be able to exchange messages with the particular physical interactor that is being modelled. We discuss next how this is achieved.

The agent/avatar (entity) model holds information about the agent/avatar as well as about interactor models. This information includes the enumeration of all the unique identifiers for all of the interactors that belong to the particular agent/avatar body as acquired from the interactor registration metadata. The set of entity models in the middleware will be referred to as the “entity model repository”.

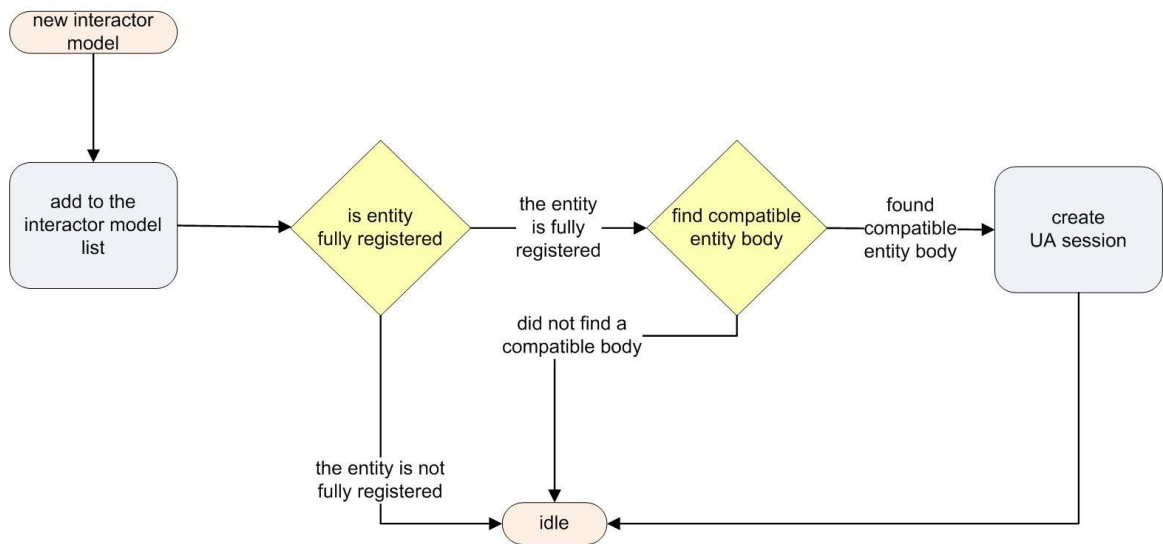
Following Flowchart 1, when the middleware processes the metadata for an interactor that has been registered, it will create two models if they do not already exist (avoiding duplicate registrations): one that represents internally the agent/avatar it belongs to and one that represents the particular interactor. For every consequent interactor registration the middleware will be associating a new interactor model with the existing agent/avatar model without having to create the same entity model multiple times.

The middleware considers an agent or an avatar to be fully registered if all of its sensors and actuators have been registered by sending their metadata descriptions through their individual connections. Each agent and avatar model stores a list that enumerates the identifiers of all of the sensors and actuators that constitute the body of the agent/avatar (entity) that it models. The registration metadata of every interactor includes the particular list. In practice only one registered interactor with the particular list would be enough for passing this information to the middleware. The purpose of this list is to enable the middleware to test when an entity is fully registered by checking if all entity identifiers point to interactor models that have already been created.

3.3.3. Discovery and Binding

In this section we will describe discovery in the proposed framework. As shown in Flowchart 2, every time a new interactor model is created, the middleware will check if the particular entity that it belongs to is fully registered. If it is then it will search through the entity model repository for a compatible and fully registered body. Two bodies are compatible if one of them

is attributed to an agent, the other to an avatar and all of the sensors and actuators of the agent body are compatible for binding to sensors and actuators of the avatar body. In the proposed middleware, discovery involves identifying agent and avatar bodies that consist of compatible interactors for the creation of exclusive communication sessions between them within the context of a UA session. An agent body is typically a set of service consumer and an avatar body is typically a set of service providers. Binding is the result of the discovery logic.



Flowchart 2. The discovery logic.

In the following we will describe what the aforementioned compatibility entails. An agent and a physical interactor are compatible for binding if:

- The former is the consumer and the latter the provider of a service.
- They both support the same type of binding and describe either a sensor or an actuator.
- The binding type is “targeted” and they both share the same value in the “target” section of the metadata, or the binding type is agnostic and the target value will not be evaluated.

Consider again the simple scenario of 3.1. Agnostic binding would be more appropriate if the end user wanted to replicate the same scenario with the same setting consisting of a door sensor and a light switch in a different room (the

interactors of each room are grouped into different avatars). The idea here is to register a set of agents allowing the middleware to dynamically match them to avatars in a random manner (as both rooms feature the same setting) in order to create UAs. On the other hand, if we wanted to use a single avatar controlling the sensors and actuators of both rooms then the binding type would have had to be set to “targeted”. This way we avoid undesired behaviour such as for example the situation in which input from the door sensor in one room resulting to switching the light of the other room. This would have been a possibility if the middleware performed the bindings based only on functionality and not locality (“target”).

If no compatible body is found for an entity, it will remain available for discovery in the entity model repository.

3.3.4. UA sessions

A successful binding is achieved when all the interactors of an agent are bound to interactors of an avatar enabling the creation of a “Ubiquitous Agent session”. Figure 3-5 illustrates a UA session where agent and avatar interactors exchange messages to support software agents for accessing and possibly changing the physical environment. The interaction in the UA session context takes place within the exclusive relationships between agent and avatar interactor models.

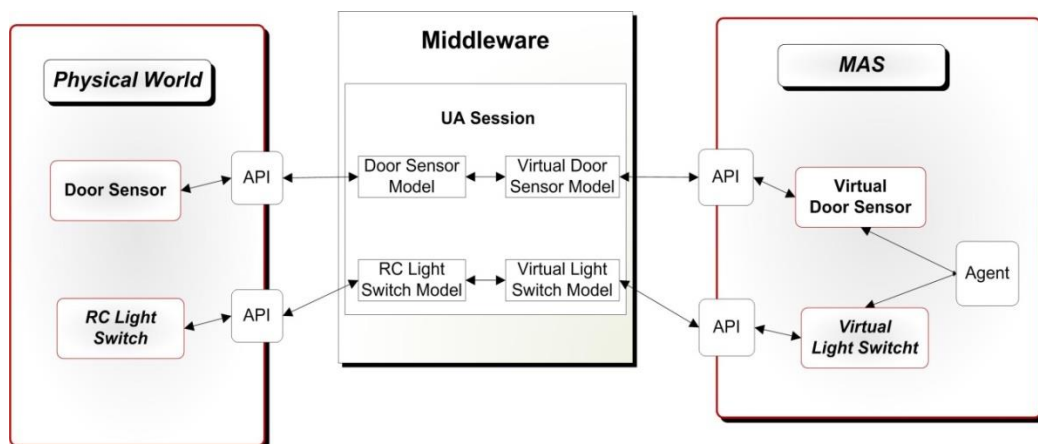


Figure 3-5. Example of a Ubiquitous Agent session.

There are three ways in which interaction is initiated when a UA is started: (a) when an agent is trying to access the environment through active observation (message from agent to avatar) (b) when an avatar detects environment change and creates a passive observation for the agent (the agent did not actively look for it - e.g. consider a fire alarm in a building) and (c) when an agent requests for an action to be executed. The aforementioned types of observations are discussed in the KGP model, see [3]. The UA acts as a mediator [21] achieving loose coupling. For example in the scenario of 3.1, the door sensor would be sending messages to the middleware containing its status changes and the UA will ensure that the agent “door” sensor will be receiving them. The agent “door” sensor will then pass them to the decision making component of the GOLEM agent. The latter would be deciding whether the agent “light” switch would send a control message to the middleware in order for it to be received by the wireless light switch which in its turn would switch the light on or off according to the received command.

Another requirement for the middleware is to monitor the state of the connections involved in a UA session. The approach should also provide with resilience and failover mechanisms in the case that something has gone wrong with the middleware itself. Having described the requirements and what the proposed middleware should do, it is time to answer the question of how to do this by presenting the architecture of the proposed middleware.

3.4. The Middleware Architecture

The middleware architecture follows SOA principles. In chapter 2 we saw that SOA can be deployed in a variety of middleware architectures. Our application requires asynchronous communication and loose coupling among the communicating entities. The main reason is that the behaviour of individual entities (agent, avatar sensors/actuators) can vary depending on their design, runtime usage and the workload or time required performing a task. We

propose a message oriented middleware architecture based on SOA. In Figure 3-6 we see an overview of the proposed architecture for the middleware.

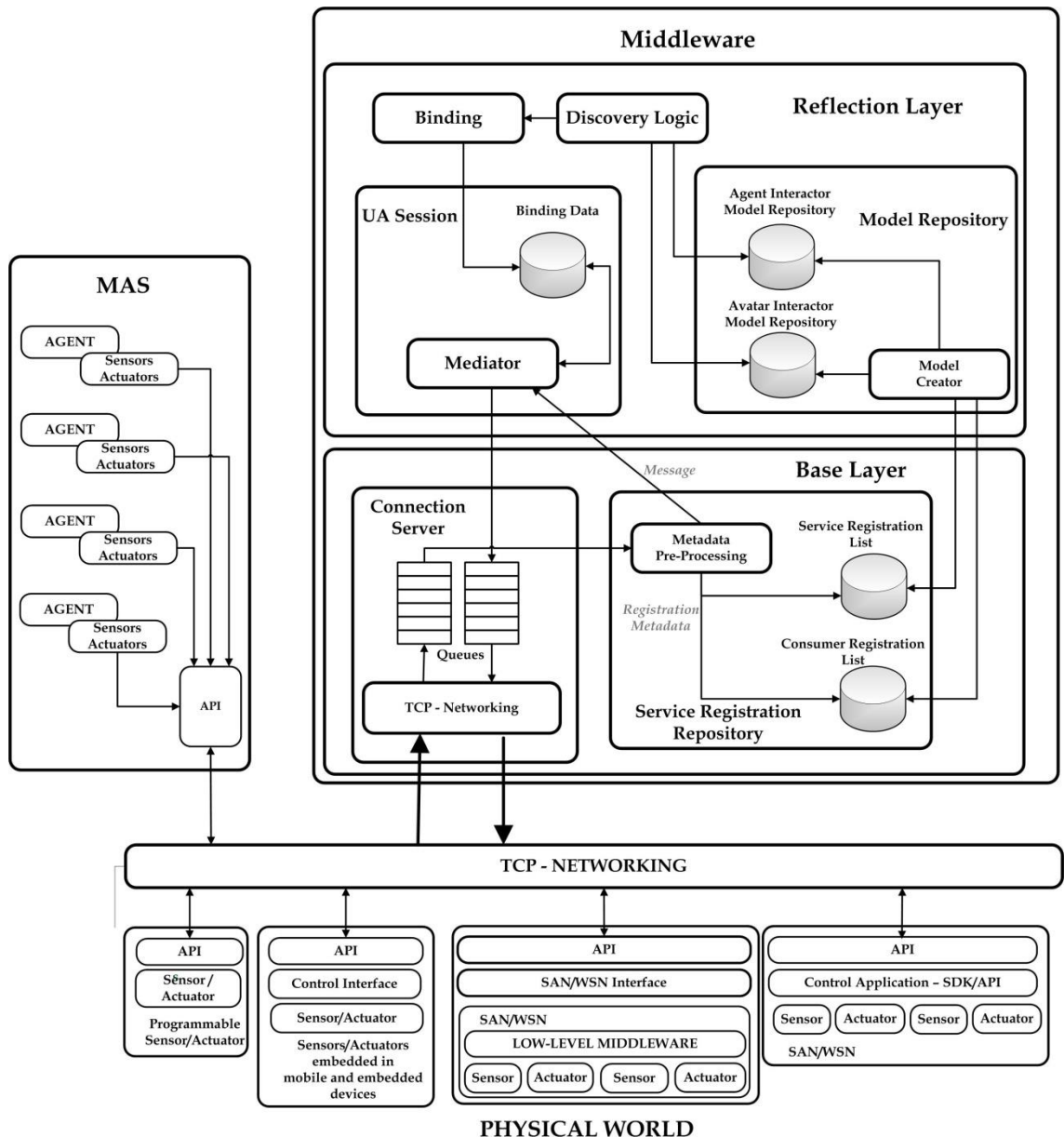


Figure 3-6. The architecture of the proposed middleware.

The proposed middleware uses an integration broker that implements reflection. The reflection functionality creates models of agents, avatars and their interactors based on context from SOA services and reasons on these models in order to implement UAs as per the requirements of the previous section. The main components of the middleware are implemented in two

layers: the base layer and the reflection layer that adds more sophisticated functionality to the system.

The base layer of the middleware uses transport layer protocols (TCP [49]) to implement a connection server that manages connections from networked physical/agent interactors that use the middleware API. It also creates an incoming and an outgoing queue for every connection. Before a message is handled, it should be placed in an incoming queue and similarly, outgoing messages towards interactors are placed in outgoing queues.

The interactors register to the middleware as service providers or as service consumers using their service metadata descriptions. The “service registration” component deals with acquiring and processing the interactor metadata descriptions from the connection server and storing them in a “service registration repository” data storage component in the base layer of the middleware. All the services will be available via the middleware and consumers will register interest for services through it. This way we implement the agent/avatar registration requirement of the previous section.

Returning to the scenario of 3.1, the light switch actuator would be described as a service provider. It would advertise the service via the middleware by sending its service description in the form of descriptive metadata. On the other hand, the agent “light” actuator would register by sending a description of the service that it would require to consume. This service would again use metadata to describe the light switch functionality. As we will see in the following, the communication between the provider and the consumer of the service begins if the middleware binds the agent to the avatar body (binds all of their respective interactors) and creates a UA session for them.

The middleware should perform “store and forward” functionality in order to pass messages from the providers to the consumers and vice versa.

The reflection layer provides four main functionalities: the creation of models as described in the requirements section, the discovery of compatible agent and avatar bodies (see 3.3) and the binding between them via their models, the management of the resulting UAs and the monitoring of the communications

for resilience. The components that implement these functionalities are referenced in Figure 3-6.

The reflection layer uses the service registration repository to create interactor, agent and avatar models. The interactor models are causally connected with the base interactors they represent. They collectively constitute a self-representation of the system. The causal connections between base entities and models are implemented via the service oriented architecture that enables communication between them. Changes in the state of the base entities should reflect to changes in self-representation, and vice versa.

The idea is to manipulate and reason on the models to solve problems regarding the base-layer. The models contain descriptions of the interactors that they represent and the discovery component of the architecture uses these descriptions in order to find compatible bodies and aggregate them into UAs as per the requirements of 3.3.3. The “Ubiquitous Agent” component (UA Session in Figure 3-6) is responsible for managing the communication within a UA context as per 3.3.4.

From the agent/avatar interactor perspective, an API (Application Programming Interface) enables distributed interactor software to interact with the middleware. The API enables the conversion of information from the application's format to the common format of the communication (XML based SOA) within the middleware and the opposite. The API provides with the means to describe an interactor as a service or a service consumer and to also support the communication protocols for interacting with the integration broker.

Depending on the MAS implementation, the API is used directly by the interactor software, or indirectly via a separate MAS software object (e.g. GOLEM objects [82]). In the second case the sensors/actuators interact with the MAS object and the latter translates this interaction to API function calls. In the physical world layer the API is used as per the description of section 3.2.2.

3.5. Framework Specification

The following chapter presents a formal specification of the proposed in this thesis approach and solution. Formal methods are concerned with the use of mathematical techniques and notations for describing, analysing and specifying the properties of software systems (Clarke et al. [20]). Typically, the requirements of a system are described using informal language. The specification of the system is essentially the transformation of the requirements into what the system must do without saying how it is to be done [45].

The Z-notation [45] will be used to specify the properties of our proposed system. The Z-notation is based on Zermelo-Fraenkel set theory and first order predicate logic. It is commonly used to specify and analyse the behaviour of systems and to model it. The Z-notation was chosen for the following reasons:

- It is a formal “language”. Its resemblance to a programming language and use of set theory make it accessible by a wider audience from programming and mathematics backgrounds [45].
- The abstraction provided by the Z-notation formal specification allows us to describe the functionality of the system without having to refer to huge amounts of program code or assumptions regarding imprecise documentations and descriptions [45].

A Z specification consists of schemas, each representing a part of the system. Schemas describe static and dynamic aspects of a system [45] such as: the states it can occupy, the invariant relationships that are maintained as the system evolves from state to state, the possible operations including the relationship between their inputs and outputs and dynamic state changes. Z schemas are divided into two sections, the upper section for declaring variables as well as their types and the lower section that creates relationships using the variables of the first while putting constraints on them. The specification of chapter 4 is presented in Z. The specification uses diagrams to represent the schemas such as the state space schema and the operation schema.

In order to describe the Z-Notation we will use Z-notation schemas to specify a system that describes the room in the scenario of 3.1. Sets are fundamental in Z. In our example we define the “SENSORS” and the “SWITCHES” sets as the sets of all sensors and switches in a room:

[SWITCHES, SENSORS]

We use a schema to represent a room consisting of a finite set of sensors and switches that have joined it. A room has a door sensor (“door_sensor”) and a remote controlled light switch (“rc_light_switch”).

Room
switches : \mathbb{P} SWITCHES
sensors : \mathbb{P} SENSORS
door_sensor: SENSORS
rc_light_switch: SWITCHES
door_sensor \in sensors
rc_light_switch \in switches

We are describing the state space of a system and the variables such as the “sensors”, “switches”, “door_sensor” and the “rc_light_sensor” represent important observations which we can make of the state. The predicate section presents a constraint that should always be fulfilled which is that the door_sensor belongs to the set of the sensors in the room and similarly the rc_light_switch is one of the switches of the room. We can use schemas to define new variables e.g. room_1, room_2: Room. We access variables in the schema by using the syntax “schema name ‘.’ variable name” e.g. room_1.rc_light_switch.

In this description of the state space of the system, we have not been forced to place a limit on the number of items recorded in “Room”, nor to say that the entries will be stored in a particular order. We have avoided making a premature decision for example about the format of the “sensors” or the “switches”. At this stage we treat them as abstract objects.

The “operation schemas” describe operations on the variables of schemas. In general, operations cause changes in variables of the particular schema. The following operation specifies the addition of a new sensor to the “Room”.

AddMotionSensor
Δ Room
motion_sensor?: SENSORS
sensors = sensors \cup {motion_sensor?}

We add the new motion sensor “motion_sensor?” to the set of sensors: “sensors = sensors \cup {motion_sensor?}”. The use of the set union symbol ‘ \cup ’ denotes that the “motion_sensor?” sensor is now a member of the “sensors” set and therefore a sensor in the room. The ‘ Δ ’ symbol is used before the name of the schema upon which the operation will take place and denotes that the state of the system has changed after this operation. If the state had not been changed, the ‘ Ξ ’ symbol would have been used instead. Operations may have input variables represented using the ‘?’ symbol (see “motion_sensor?” above) and output variables using the ‘!’ symbol. We can see above that not all variables that describe the state of the “Room” system have changed with the “AddMotionSensor” operation. For example the “switches” set doesn't change when a new sensor is added. If we wanted to add a new switch we would use the following operation that specifies the addition of a new dimmer switch.

AddDimmerSwitch
Δ Room
dimmer_switch?: SWITCHES
switches = switches \cup {dimmer_switch?}

In Z we can combine schemas. In particular we can include a schema within another schema. For example the room has a subset of switches that feature network connectivity and following the scenario of 3.1 they could participate in a UA architecture.

ConnectedSwitches
Room
connected_switches: \mathbb{P} SWITCHES
connected_switches \subseteq switches

The “Room” schema is included within the “ConnectedSwitches” schema with all the variables and constraints of the “Room” being included in the corresponding sections of the “ConnectedSwitches” schema. It is equivalent to:

ConnectedSwitches
switches : \mathbb{P} SWITCHES
sensors : \mathbb{P} SENSORS
door_sensor: SENSORS
rc_light_switch: SWITCHES
connected_switches: \mathbb{P} SWITCHES
door_sensor \in sensors
rc_light_switch \in switches
connected_switches \subseteq switches

Z also allows global variables and functions (see Table 3-1) that can be defined outside schemas and within the axiomatic definitions that will be described below. The following example of an axiomatic definition describes a global variable representing a light sensor (no predicates therefore one section only):

$\left| \textit{light_sensor} : \textit{SENSORS}$

Finally, functions too (see Table 3-1) can be defined within axiomatic definitions. Axiomatic definitions may also contain two sections, one for declaring variables and one for predicates. All of their variables and predicates are global elements that can be used by other schemas. Below we see a table describing the symbols that we used for the definition of the proposed in this thesis framework.

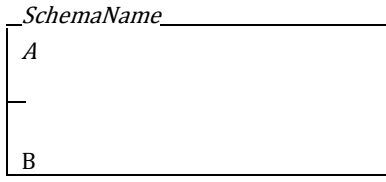

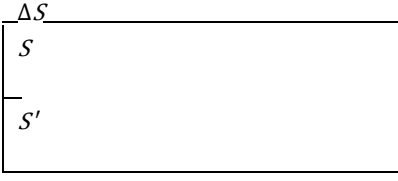
Specifications	Sets
<p>Schema box:</p>  <p>Axiomatic Definition:</p>  <p>Operation:</p>  <p>[A] basic type definition ΔS change of state $\exists S$ no change of state</p> <p>Logical operators</p> <p>\neg negation \wedge AND, conjunction \vee OR, disjunction $P \Rightarrow Q$ implication $P \Leftrightarrow Q$ equivalence $\forall x : T$ for all $x : T$, universal quantifier $\exists x : T$ there exists $x : T$, existential quantifier $\exists_1 x : T$ exactly one $x : T$, unique quantifier</p>	<p>$x \in S$ membership $x \notin S$ non-membership $\{x_1, \dots, x_n\}$ set $S \subseteq T$ subset relation $S \subset T$ proper subset relation \emptyset empty set \cup set union \cap set intersection \setminus set difference $A : \mathbb{P} B$ declares A to be a subset of B \bullet used for quantification</p> <p>Functions \ Relations</p> <p>$A \mapsto B$ partial function, a function that maps a subset of its domain type. $\text{dom } X$ domain of relation. $A \rightarrow B$ total function, a function that maps every element of its domain type.</p> <p>Basic Expressions</p> <p>$=$ equality \neq inequality $\#$ number of elements in finite set if P then E1 conditional expression else E2 $\forall x : X \mid P_1 \bullet P_2$ universal quantification any element of X that satisfies predicate P1 also satisfies predicate P2. $\exists x : X \bullet P$ existential quantification, at least one element of X satisfies predicate P. $\exists x : X \bullet P$ existential quantification, exactly one element of X satisfies predicate P.</p>

Table 3-1. Z-notation sample reference table.

In chapter 4 we can see the schemas that were used for the specification of the system that is proposed in this thesis. The specification that is described in chapter 4 has been type-checked using fuzz for Z [48].

3.6. Summary

In this chapter we provided a general overview of the architecture for the proposed in this thesis system. We saw the main components and the general requirements for each in order to participate in the proposed architecture. Based on the requirements and the architecture of the system, we may proceed with the following chapter that uses the Z-Notation to provide with formal specification of a framework for it.

4. THE UBIQUITOUS AGENTS FRAMEWORK

In this chapter we specify using the Z-notation the middleware that we introduced in the previous chapter. The resulting framework is intended to be used as a reference for implementing middleware for UAs. In section 4.1 we will specify the middleware and how the subsystems (MAS, physical world) can use it in order to create UAs. Then in section 4.2 we specify an API (Application Programming Interface) that is used by distributed interactor software (which we will be calling DIS for the remainder of this chapter) in order to interact with the middleware. The importance of the formal specification is that the resulting framework can be transparent, intelligible and easy to use as the foundation for the creation of systems and platforms that share similar aims and objectives to the ones of this thesis. We used the fuzz type-checker [48] to check the specification of the framework for inconsistencies and ambiguities.

4.1. Specifying the Middleware

As we saw in the previous chapter the middleware can be described as a set of UA sessions. The basic framework for the middleware supports the following functionality:

- a) Connecting and registering agent/avatar interactor software. The distributed interactor software (DIS) will be characterized with internal representations that we will be calling models.
- b) Binding of agent to avatar bodies (by linking their interactors). This is the result of the discovery logic that identifies compatible agent and avatar bodies (see 3.3.3 and 4.1.7). As we have seen in the requirements that were set in the previous chapter this entails the processing of the models

that were created in a). It essentially binds models and the communication between the actual agent and avatar interactors will take place via these models, thus implementing a UA session.

c) Managing the interaction within the UA session.

This way we specify the middleware as a set of UAs. We will use the requirements of the previous chapter to specify the middleware functionality as a series of Z-Notation schemas and operations. For the purposes of the middleware section of the framework, an interactor will be referring to sensor or actuator software that uses an API to interact with the middleware (see chapter 3). We assume that the reader is familiar with the requirements and the architecture of the middleware as described in chapter 3.

4.1.1. Middleware definition

We begin by describing the finite sets that represent the basic types of the specification that are used to describe the main aspects of the system.

[ENTITY_MODELS, INTERACTOR_MODELS, UBIQUITOUS_AGENTS,
METADATA]

Their descriptions:

- ENTITY_MODELS: this set is representing the agent and avatar models as defined in the reflection layer of the middleware system (see section 3.3.2).
- INTERACTOR_MODELS: the set of interactor models in the reflection layer of the middleware system (see section 3.3.2).
- UBIQUITOUS_AGENTS: the set representing all Ubiquitous Agents in the system. A UA is a communication session between agent and avatar interactors.
- METADATA: a set of metadata describing agents, avatars and interactors.

Now we can define the state space of the middleware with a schema:

```
Middleware
-----
entities :  $\mathbb{P}$  ENTITY_MODELS
interactors :  $\mathbb{P}$  INTERACTOR_MODELS
UA_sessions:  $\mathbb{P}$  UBIQUITOUS_AGENTS
metadata:  $\mathbb{P}$  METADATA
```

Listing 4-1. The state space of the middleware system.

Besides the basic types of the middleware system we define more sets that will allow us to specify the functionality of the proposed framework. These sets describe the data types that are used by the middleware (their descriptions can be found in appendix A-1):

```
[INTERACTOR_IDENTIFIERS, SERVICE_DESCRIPTIONS, TARGET, MESSAGES,
ENTITY_IDENTIFIERS]
```

We also define global variables representing the above sets in the middleware system.

```
interactorIDs:  $\mathbb{P}$  INTERACTOR_IDENTIFIERS
entityIDs:  $\mathbb{P}$  ENTITY_IDENTIFIERS
srv_descriptions:  $\mathbb{P}$  SERVICE_DESCRIPTIONS
target:  $\mathbb{P}$ TARGET
message:  $\mathbb{P}$ MESSAGES
```

Listing 4-2. Creating global variables for sets that describe elements of the middleware.

Finally, we define free types which are sets containing exact numbers of predefined values (their descriptions can be found in Appendix A-1):

```

OUTCOME ::= ok | fail
BINDING_TYPE ::= targeted | agnostic
TYPE_OF_INTERACTOR ::= sensor | actuator
ENTITY_TYPE ::= agent | avatar
SERVICE_TYPE ::= provider | consumer

```

In the previous chapter we saw that the middleware follows a two layered architecture with a base and a reflection layer. In order to specify the operations of the Ubiquitous Agents middleware we need to first define functions upon the above sets. As we will see next, the functions represent how the types that are described by the above sets are interrelated within the middleware and in particular how the reflection layer uses the base-layer to implement its functionality.

4.1.2. Interactors as Service Providers and Consumers

DIS register to the middleware as service providers and service consumers by sending metadata descriptions to the middleware consisting of information that contains all the elements that were described in section 3.3.2. We specify interactor descriptions using Z:

```

BS_IncomingRegistrationData
interID?: INTERACTOR_IDENTIFIERS
inter_type?: TYPE_OF_INTERACTOR
bindingType?: BINDING_TYPE
locTarget?: TARGET
entityID?: ENTITY_IDENTIFIERS
entityType?: ENTITY_TYPE
expectedInteractorIDs?: PINTERACTOR_IDENTIFIERS
serv_type?:SERVICE_TYPE
service_description?: SERVICE_DESCRIPTIONS

```

Listing 4-3. Incoming metadata description from an interactor.

The “serv_type?” specifies whether the interactor is a service provider or a consumer. Most of the names here are self-explanatory and appendix A-1

describes the types of the above variables. For example the “service_description?” variable is of type “SERVICE_DESCRIPTIONS” and describes the functionality of an interactor as a service provider or consumer. What needs explaining is the “locTarget” that stores the target value that is used to map (bind) an interactor to another interactor with the same target value. Also the “expectedInteractorIDs” variable stores the set of interactor identifiers that point to the interactors that are expected to register and belong to the same agent or avatar body.

4.1.3. Base Layer Functionality

The middleware accepts registrations by agent and avatar distributed interactor software. We use the following axiomatic definitions to specify global functions that formally define the base layer functionality of the middleware that enables it to manage registration and store and process the registration data. We specify the following base layer functions:

```

registration_list: INTERACTOR_IDENTIFIERS → METADATA
reg_id_of_interactor: METADATA → INTERACTOR_IDENTIFIERS
reg_type_of_interactor: METADATA → TYPE_OF_INTERACTOR
reg_binding_type_of_interactor: METADATA → BINDING_TYPE
reg_target_of_interactor: METADATA → TARGET
reg_id_of_entity: METADATA → ENTITY_IDENTIFIERS
reg_type_of_entity: METADATA → ENTITY_TYPE
reg_expected_interactorIDs: METADATA → PINTERACTOR_IDENTIFIERS
reg_service_type_of_interactor: METADATA → SERVICE_TYPE
reg_description_of_interactor : METADATA → SERVICE_DESCRIPTIONS

```

Listing 4-4. Base layer functions.

The functions specify base layer functionality that was described in section 3.3, e.g. the “registration_list” function specifies the service registration list. We use the above functions to specify the operation for receiving and processing the registration descriptions from the DIS. The main idea is that when the middleware receives registration data as specified in “BS_IncomingRegistrationData”, it will first check whether the interactor has

already been registered using the “registration_list” function. If it has not, it will then use the rest of the functions that were specified in Listing 4-4 (e.g. “reg_id_of_interactor”, “reg_type_of_interactor” etc.) in order to store the information from “BS_IncomingRegistrationData” in a new metadata entry (see “reg_data!” below). It will then add this registration metadata to the “registration_list” for the particular interactor identifier. The functions of Listing 4-4 are consequently used to access information from the metadata entries.

BS_ReceiveAndProcessRegistrationData BS_IncomingRegistrationData reg_data!: METADATA
$\text{registration_list}(\text{interID?}) \in \text{METADATA} \wedge$ $\text{reg_id_of_interactor} = \text{reg_id_of_interactor} \cup \{\text{reg_data!} \mapsto \text{interID?}\} \wedge$ $\text{reg_type_of_interactor} = \text{reg_type_of_interactor} \cup \{\text{reg_data!} \mapsto \text{inter_type?}\} \wedge$ $\text{reg_binding_type_of_interactor} = \text{reg_binding_type_of_interactor} \cup$ $\{\text{reg_data!} \mapsto \text{bindingType?}\} \wedge$ $\text{reg_target_of_interactor} = \text{reg_target_of_interactor} \cup \{\text{reg_data!} \mapsto \text{locTarget?}\} \wedge$ $\text{reg_id_of_entity} = \text{reg_id_of_entity} \cup \{\text{reg_data!} \mapsto \text{entityID?}\} \wedge$ $\text{reg_expected_interactorIDs} = \text{reg_expected_interactorIDs} \cup$ $\{\text{reg_data!} \mapsto \text{expectedInteractorIDs?}\} \wedge$ $\text{reg_type_of_entity} = \text{reg_type_of_entity} \cup \{\text{reg_data!} \mapsto \text{entityType?}\} \wedge$ $\text{reg_service_type_of_interactor} = \text{reg_service_type_of_interactor}$ $\cup \{\text{reg_data!} \mapsto \text{serv_type?}\} \wedge$ $\text{reg_description_of_interactor} = \text{reg_description_of_interactor} \cup$ $\{\text{reg_data!} \mapsto \text{service_description?}\} \wedge$ $\text{registration_list} = \text{registration_list} \cup \{\text{interID?} \mapsto \text{reg_data!}\}$

Listing 4-5. Specifying the receiving and processing of registration data by the middleware.

4.1.4. The Entity Models in the Reflection Layer

According to the requirements that were laid in the previous chapter, when the middleware processes the metadata for the first interactor that has been registered, it will create two models: one that represents internally the agent/avatar and one that represents the particular interactor. For every consequent interactor registration the middleware will be associating a new interactor model to the existing agent/avatar model. We specify the following

functions that belong to the reflection layer of the middleware and enable the framework to specify the operation for creating a new entity model:

```

ent_model_model_list: ENTITY_IDENTIFIERS  $\rightarrow$  ENTITY_MODELS
ent_model_metadata: ENTITY_MODELS  $\rightarrow$  METADATA
ent_model_id: METADATA  $\rightarrow$  ENTITY_IDENTIFIERS
ent_model_type: METADATA  $\rightarrow$  ENTITY_TYPE
ent_model_expected_ids: METADATA  $\rightarrow$   $\mathbb{P}$ INTERACTOR_IDENTIFIERS
ent_model_registered_ids: METADATA  $\rightarrow$   $\mathbb{P}$ INTERACTOR_IDENTIFIERS

```

Listing 4-6. Reflection layer functions for the creation of entity models.

We saw in 3.3.2 that entity models contain metadata. The framework specifies the above functions in order to acquire information from the registration list of the base layer using the functions of Listing 4-4. It will then use this information to create an entity model and add metadata to it (“ent_model_metadata” function). This information includes an identifier for the entity, its type (agent or avatar) and a list of all the unique identifiers for all of the sensors and actuators that constitute the body of the particular entity. The “ent_model_model_list” function is implementing the “entity model repository” (see requirements in previous chapter). We formally define the more complex global function that uses the above functions to create the metadata of a particular entity model based on the registration metadata of the base layer as accessed using the functions of Listing 4-4:

```

ent_model_create_metadata: METADATA  $\rightarrow$  METADATA
 $\exists$ registration_data: METADATA; ent_model_metadata: METADATA
•
ent_model_create_metadata(registration_data) = ent_model_metadata  $\Leftrightarrow$ 
ent_model_id = ent_model_id  $\cup$ 
{ ent_model_metadata  $\mapsto$  reg_id_of_entity(registration_data) }  $\wedge$ 
ent_model_type = ent_model_type  $\cup$ 
{ ent_model_metadata  $\mapsto$  reg_type_of_entity(registration_data) }  $\wedge$ 
ent_model_expected_ids = ent_model_expected_ids  $\cup$ 
{ ent_model_metadata  $\mapsto$  reg_expected_interactorIDs (registration_data) }

```

Listing 4-7. Creating the entity model metadata.

According to the framework, the entity model (as we will also see in the following) is created when the first interactor is registered. For consequent interactor registrations to the base layer we only update the list of registered interactor identifiers in the metadata of the entity model (see “ent_model_registered_ids” in Listing 4-8). The rest of the metadata have already been added with the registration of the first interactor. We specify the updating of the metadata layer of an entity model with the following global function:

ent_model_update_metadata: METADATA \rightarrow METADATA \exists reg_metadataIn: METADATA; model_metadataOut: METADATA; registeredIDs: \mathbb{P} INTERACTOR_IDENTIFIERS; entID: ENTITY_IDENTIFIERS • ent_model_update_metadata(reg_metadataIn) = model_metadataOut \Leftrightarrow entID = reg_id_of_entity(reg_metadataIn) \wedge model_metadataOut = ent_model_metadata(ent_model_model_list(entID)) \wedge registeredIDs = ent_model_registered_ids(model_metadataOut) \wedge registeredIDs = registeredIDs \cup { reg_id_of_interactor(reg_metadataIn) } \wedge ent_model_registered_ids = ent_model_registered_ids \cup { model_metadataOut \mapsto registeredIDs }

Listing 4-8. Updating the metadata of an entity model.

The next function that we will specify is used to return the entity model of a newly registered interactor and create it if it does not already exist. The input is the registration metadata from the base layer for the particular interactor. It will use the entity identifier for the particular entity to check whether the entity model exists in the entity model repository using the “ent_model_model_list” function. If it does not exist it will create and update it with all the relevant metadata. Otherwise it returns the existing entity model after it has updated the list of registered interactor identifiers. In the first case it calls the “ent_model_create_metadata” function of Listing 4-7 and in the second the “ent_model_update_metadata” of Listing 4-8.

```

ent_model_get_model: (METADATA × ℙ ENTITY_MODELS)
→ ENTITY_MODELS
-----
∃_1 registration_data: METADATA; entMetadata: METADATA;
entities: ℙ ENTITY_MODELS; entityModel: ENTITY_MODELS •
(ent_model_get_model(registration_data, entities) = entityModel ⇔
ent_model_model_list (reg_id_of_entity(registration_data)) ∈ ENTITY_MODELS ∧
entities = entities ∪ {entityModel} ∧
entMetadata = ent_model_create_metadata(registration_data) ∧
ent_model_model_list = ent_model_model_list ∪
{ reg_id_of_entity(registration_data) ↦ entityModel } ∧
ent_model_metadata = ent_model_metadata ∪ { entityModel ↦ entMetadata } ∧
entMetadata = ent_model_update_metadata(registration_data) ) ∨
(ent_model_get_model(registration_data, entities) =
ent_model_model_list (reg_id_of_entity(registration_data)) ⇔
entMetadata = ent_model_update_metadata(registration_data))

```

Listing 4-9. Return the entity model and create one if it does not exist.

Having specified the metadata of the entity model, we specify the function for associating a set of interactor models to it:

```

ent_model_registered_interactors: ENTITY_MODELS → ℙ INTERACTOR_MODELS

```

Listing 4-10. Adding an interactor model to an entity model.

4.1.5. The Interactor Models in the Reflection Layer

Interactor models have a metadata and an interactivity layer (see 3.3.2). The metadata layer is updated using functions of the reflection layer specified as:

```

intr_model_by_id: INTERACTOR_IDENTIFIERS → INTERACTOR_MODELS
intr_model_metadata: INTERACTOR_MODELS → METADATA
intr_model_id: METADATA → INTERACTOR_IDENTIFIERS
intr_model_type : METADATA → TYPE_OF_INTERACTOR
intr_model_service_type: METADATA → SERVICE_TYPE
intr_model_description: METADATA → SERVICE_DESCRIPTIONS
intr_model_binding_type: METADATA → BINDING_TYPE
intr_model_target: METADATA → TARGET
intr_model_entityID: METADATA → ENTITY_IDENTIFIERS

```

Listing 4-11. Reflection layer functions for the creation of interactor models.

The following function uses the above functions of the reflection layer and the registration data as stored in the registration list of the base layer of the middleware in order to create the interactor model and add its metadata.

<pre> CreateInteractorModel: (METADATA×PINTERACTOR_MODELS) →INTERACTOR_MODELS ----- ∃ registration_data, interactorMetadata: METADATA; interactors:PINTERACTOR_MODELS; newInteractorModel:INTERACTOR_MODELS • CreateInteractorModel(registration_data, interactors)= newInteractorModel ⇔ intr_model_by_id(reg_id_of_interactor(registration_data)) ∈ INTERACTOR_MODELS∧ interactors = interactors ∪{ newInteractorModel}∧ intr_model_id = intr_model_id ∪ { interactorMetadata ↦ reg_id_of_interactor(registration_data)}∧ intr_model_type = intr_model_type ∪ { interactorMetadata ↦ reg_type_of_interactor (registration_data)}∧ intr_model_service_type = intr_model_service_type ∪ { interactorMetadata ↦ reg_service_type_of_interactor (registration_data)}∧ intr_model_description = intr_model_description ∪ { interactorMetadata ↦ reg_description_of_interactor (registration_data)}∧ intr_model_binding_type = intr_model_binding_type ∪ { interactorMetadata ↦ reg_binding_type_of_interactor (registration_data)}∧ intr_model_target = intr_model_target ∪{ interactorMetadata ↦ reg_target_of_interactor (registration_data)}∧ intr_model_entityID = intr_model_entityID ∪{ interactorMetadata ↦ reg_id_of_entity (registration_data)}∧ intr_model_metadata = intr_model_metadata∪{ newInteractorModel ↦interactorMetadata} </pre>

Listing 4-12. Creating a new interactor model while also adding the metadata.

When registration is complete, the base layer will be using two message queues to serve the message exchanges between the middleware and the interactor. The first will be used for the incoming messages from the particular interactor (and only) and the second for the messages from the middleware towards it.

<pre> incoming_queue : INTERACTOR_IDENTIFIERS ↔ MESSAGES outgoing_queue : INTERACTOR_IDENTIFIERS ↔ MESSAGES </pre>
--

Listing 4-13. The incoming and outgoing queues for the particular interactor identifier.

These queues are used as the interactivity layer by the interactor model with the particular identifier. A message from the agent or physical interactor will reach the incoming message queue that is accessed by the corresponding interactor model. The messages that are placed in the outgoing queue originate from the incoming queues of other interactor models. These messages are subsequently sent to an agent or physical interactor.

Finally, after the middleware has created a new interactor model it will associate it with the corresponding entity model. We specify this with the following function.

$\text{addInteractorToEntity}: (\text{ENTITY_MODELS} \times \text{INTERACTOR_MODELS}) \rightarrow \mathbb{P}\text{INTERACTOR_IDENTIFIERS}$
$\exists \text{interactorModel}: \text{INTERACTOR_MODELS}; \text{entityModel}: \text{ENTITY_MODELS};$
$\text{interactorIDsOfEntity}: \mathbb{P}\text{INTERACTOR_IDENTIFIERS};$
$\text{interactorsOfEntity}: \mathbb{P}\text{INTERACTOR_MODELS} \bullet$
$\text{addInteractorToEntity} (\text{entityModel}, \text{interactorModel}) = \text{interactorIDsOfEntity} \Leftrightarrow$
$\text{interactorsOfEntity} = \text{ent_model_registered_interactors} (\text{entityModel}) \wedge$
$\text{interactorsOfEntity} = \text{interactorsOfEntity} \cup \{ \text{interactorModel} \} \wedge$
$\text{ent_model_registered_interactors} = \text{ent_model_registered_interactors} \cup$
$\{ \text{entityModel} \mapsto \text{interactorsOfEntity} \}$

Listing 4-14. This function specifies the addition of a new interactor model to an entity model.

4.1.6. Registration

At this point we have defined the reflection and base layer functionality that would allow for the specification of the registration operation. We will use the above functions to formally specify the operation for registering an interactor.

The “BS_ReceiveAndProcessRegistrationData” (Listing 4-5) specifies how the middleware receives the registration descriptions from DIS and checks whether the interactors have already been registered. If an interactor has not registered the “CreateInteractorModel” (Listing 4-12) function will create the model for it and associate it with the appropriate entity model (see “addInteractorToEntity” in Listing 4-14). We get the entity model by calling the “ent_model_get_model”

function and if one does not already exist it will create and return it. We specify the above with the following operation:

RegisterInteractor ΔMiddleware BS_ReceiveAndProcessRegistrationData registeredBodyInteractorIDs: \mathbb{P} INTERACTOR_IDENTIFIERS <hr/> intr_model_by_id (interID?) \notin interactors; registeredBodyInteractorIDs = addInteractorToEntity(ent_model_get_model (reg_data!, entities) , CreateInteractorModel(reg_data! ,interactors))

Listing 4-15. Interactor registration

4.1.7. Reflective Discovery & Binding

We call the binding between service providing and consuming interactors reflective because it takes place in the reflection layer using models that reflect their functionality within the middleware. The “Bind” operation is specified in the end of this section. This operation uses a number of functions that will need to be specified before we present it. It is the result of the discovery logic that will also be described in this section.

Part of the discovery functionality is to evaluate agent and avatar models by performing an initial check to ensure that one entity is an agent, the other entity is an avatar and they are both fully registered. We remind that agents and avatars are considered to be fully connected when all of their sensors and actuators have been registered. We formally specify this with the following function that determines that an entity is fully registered by comparing the sets returned by the “ent_model_registered_ids” and “ent_model_expected_ids” functions (described in 4.1.4). If they are the same it means that all the interactors of a particular entity have already registered. If all above statements are true the function will return “ok” otherwise it will return fail.

$\text{initDiscoveryChecks}:(\text{ENTITY_MODELS} \times \text{ENTITY_MODELS}) \rightarrow \text{OUTCOME}$ $\exists ! \text{firstEntityModel, secondEntityModel}:\text{ENTITY_MODELS} \bullet$ $(\text{initDiscoveryChecks}(\text{firstEntityModel}, \text{secondEntityModel}) = \text{ok} \Leftrightarrow$ $\text{ent_model_type}(\text{ent_model_metadata}(\text{firstEntityModel})) \neq$ $\text{ent_model_type}(\text{ent_model_metadata}(\text{secondEntityModel}))$ $\wedge \text{ent_model_registered_ids}(\text{ent_model_metadata}(\text{firstEntityModel})) =$ $\text{ent_model_expected_ids}(\text{ent_model_metadata}(\text{firstEntityModel}))$ $\wedge \text{ent_model_registered_ids}(\text{ent_model_metadata}(\text{secondEntityModel})) =$ $\text{ent_model_expected_ids}(\text{ent_model_metadata}(\text{secondEntityModel})))$ $\vee \text{initDiscoveryChecks}(\text{firstEntityModel}, \text{secondEntityModel}) = \text{fail}$
--

Listing 4-16. Check whether to attempt a binding.

A UA entails the exclusive communication of every interactor in an agent body with an interactor in an avatar body for control or sensing. We call “reflective” binding the dynamic creation of such relationships by the middleware at runtime. Before we describe the process of binding we need to formally specify the following function:

$\text{bound} : \text{INTERACTOR_MODELS} \rightarrow \text{INTERACTOR_MODELS}$
--

Listing 4-17. Support function for binding.

The “bound” function specifies a relationship between two interactor models. When it is applied to an interactor model it returns the interactor model with which it has formed a binding. We use the above function to formally specify how the middleware implements the binding between two interactors.

$\text{createBinding}:(\text{INTERACTOR_MODELS} \times \text{INTERACTOR_MODELS}) \rightarrow \text{OUTCOME}$ $\exists \text{InteractorF, InteractorS}:\text{INTERACTOR_MODELS} \bullet$ $\text{createBinding}(\text{InteractorF}, \text{InteractorS}) = \text{ok} \Leftrightarrow$ $\text{bound} = \text{bound} \cup \{\text{InteractorF} \mapsto \text{InteractorS}\}$ $\wedge \text{bound} = \text{bound} \cup \{\text{InteractorS} \mapsto \text{InteractorF}\}$
--

Listing 4-18. Binding two interactors.

As we saw in the requirements chapter (section 3.3), when all the interactor models of an entity are bound to interactor models of another entity, the two entities form a UA session. The following function of the “Bind” operation specifies an entity model participating in a UA session:

$$\left| \text{UA_of_entity: ENTITY_MODELS} \mapsto \text{UBIQUITOUS_AGENTS} \right.$$

Listing 4-19. A relationship between an entity model and a UA session.

We can now formally specify the part of the binding functionality that deals with the creation of the UA sessions.

$$\left| \begin{array}{l} \text{createUA: (ENTITY_MODELS} \times \text{ ENTITY_MODELS} \times \mathbb{P} \text{ UBIQUITOUS_AGENTS)} \\ \rightarrow \text{ UBIQUITOUS_AGENTS} \\ \hline \exists \text{firstEntityModel, secondEntityModel: ENTITY_MODELS;} \\ \text{UA_sessions: } \mathbb{P} \text{ UBIQUITOUS_AGENTS; UA: UBIQUITOUS_AGENTS} \bullet \\ \text{createUA}(\text{firstEntityModel, secondEntityModel, UA_sessions}) = \text{UA} \Leftrightarrow \\ \text{UA_sessions} = \text{UA_sessions} \cup \{\text{UA}\} \\ \wedge \text{UA_of_entity} = \text{UA_of_entity} \cup \\ \{\text{firstEntityModel} \mapsto \text{UA}\} \\ \wedge \text{UA_of_entity} = \text{UA_of_entity} \cup \\ \{\text{secondEntityModel} \mapsto \text{UA}\} \end{array} \right.$$

Listing 4-20. Creation of a UA.

We add the new UA session to the set of UA_Sessions. Also, when creating a new UA we also update the “UA_of_entity” relationship for both participating entities.

In order to create a binding, the middleware should implement discovery logic that identifies agents and avatars that bare compatible interactors. Two interactors are compatible when their registration descriptions that are stored in the “registration_list” of the base layer (4.1.3) fulfil a number of criteria. The first criterion is that one of them should be a description of a physical and the other one a description of an agent sensor/actuator (different entity types). The second criterion requires them to be of the same interactor type (both sensors or both actuators). The third criterion requires that the interactor service

description (as acquired by the "intr_model_description" function) that is associated with the avatar interactor model should describe the same functionality as the one required to be consumed by the agent interactor. The way this comparison is achieved is a matter of implementation as we will see in chapter 5. For the specification of our framework we will be using the term "compatible" interactor service descriptions to abstract the above functionality and we will be describing it with the equality symbol '=' e.g.:

"intr_model_description (intr_1) = intr_model_description (intr_2)".

The next criterion requires that the values returned by the "intr_model_service_type" regarding the two interactor models should not be the same i.e. one of them should be a service provider and the other one a service consumer.

The last important criterion deals with the "intr_model_binding_type". Both the agent and physical interactor models should have the same binding type. If they both support agnostic binding (the "intr_model_binding_type" function returns the value "agnostic" for both interactor models under consideration) then the criterion will be satisfied. If the value is "targeted" then they will also need to have the same value returned by the "intr_model_target" function in order to satisfy the particular discovery criterion.

We formally specify the above with the following function that specifies the discovery functionality. It discovers compatible agent and avatar models by evaluating them and their interactors and binds their interactors using "createBinding". It then creates a UA session using the "createUA" function. If the discovery and binding are successful the function will return "ok" and the new object that represents the UA session; otherwise it will return fail with an empty variable for the UA.

```

discoverAndBindBodies: (ENTITY_MODELS×ENTITY_MODELS×
 $\mathbb{P}$  UBIQUITOUS_AGENTS)→ (OUTCOME ×UBIQUITOUS_AGENTS)
 $\exists$ firstBody,secondBody:  $\mathbb{P}$ INTERACTOR_MODELS;
firstEntityModel, secondEntityModel: ENTITY_MODELS;
UAs: $\mathbb{P}$ UBIQUITOUS_AGENTS;
entMetadataF, entMetadataS:METADATA;
UA: UBIQUITOUS_AGENTS
•
discoverAndBindBodies(firstEntityModel,secondEntityModel, UAs) = (ok,UA)  $\Leftrightarrow$ 
firstBody= ent_model_registered_interactors(firstEntityModel) $\wedge$ 
secondBody= ent_model_registered_interactors(secondEntityModel) $\wedge$ 
( $\forall$  InteractorF: firstBody •( $\exists$ 1 InteractorS:secondBody•
entMetadataF = intr_model_metadata(InteractorF) $\wedge$ 
entMetadataS = intr_model_metadata(InteractorS) $\wedge$ 
intr_model_type (entMetadataF) = intr_model_type (entMetadataS) $\wedge$ 
intr_model_service_type(entMetadataF) = intr_model_service_type(entMetadataS) $\wedge$ 
intr_model_description (entMetadataF) = intr_model_description (entMetadataS) $\wedge$ 
intr_model_binding_type (entMetadataF) = intr_model_binding_type (entMetadataS) $\wedge$ 
intr_model_target (entMetadataF) = intr_model_target (entMetadataS) $\wedge$ 
createBinding(InteractorF, InteractorS) = ok) $\wedge$ 
(intr_model_binding_type(entMetadataF) = agnostic $\vee$ 
intr_model_target (entMetadataF) = intr_model_target(entMetadataS) $\wedge$ 
UA = createUA(firstEntityModel,secondEntityModel, UAs))
 $\vee$  discoverAndBindBodies(firstEntityModel,secondEntityModel, UAs) = (fail, UA))

```

Listing 4-21. Evaluates two entity bodies and binds them.

Now we have all the building blocks that will enable us to formally specify the “Bind” operation.

```

Bind
 $\Delta$ Middleware
firstEntityModel? : ENTITY_MODELS
secondEntityModel? : ENTITY_MODELS
UA! : UBIQUITOUS_AGENTS
outcome!:OUTCOME
initDiscoveryChecks(firstEntityModel?, secondEntityModel?) = ok $\wedge$ 
(outcome!,UA!) =
discoverAndBindBodies(firstEntityModel?,secondEntityModel?, UA_sessions)

```

Listing 4-22. The Bind operation.

We note that the “UA_sessions” is the set of existing ubiquitous agents in the middleware. If the function “initDiscoveryChecks” returns “ok” meaning that we can attempt the binding, the middleware will call the “discoverAndBindBodies” that will return “ok” and the new UA if the binding was successful otherwise it will return “fail”.

4.1.8. Ubiquitous Agents

The middleware can be described as a set of UAs that are the result of the binding operations. The UAs in the system are specified by the “UA_sessions” set (see Listing 4-1). In this section we will specify how messages are mediated within each UA session and in particular between the interactors that are bound to each other. The messages from the interactors of both entities that create the UA (agent and avatar) are stored in the incoming message queues of the middleware. The UA accesses these queues via their corresponding interactor models (see 4.1.5). Based on the “bound” relationship that was described in Listing 4-18 the UA will then place these messages in the outgoing queues of the interactors that are bound to them. Thus the mediated messages will be sent to the appropriate DIS.

This way the mediator is implemented in the framework. The interactor models of entity models that belong to a UA session do not exchange messages directly but through the UA session (using the “UA_of_entity” and “bound” functions). The schema below describes the “MediateMessages” operation for routing messages within a UA by reading the messages in the incoming queues of all participating interactors and placing them in the outgoing queues of the interactors that are bound to them.

MediateMessages <hr/> $\Delta\text{Middleware}$ $\text{UA?: UBIQUITOUS_AGENTS}$ <hr/> $\forall \text{senderEntityModel:ENTITY_MODELS} \bullet$ $\text{UA?} \in \text{UA_sessions} \wedge$ $\text{UA?} = \text{UA_of_entity}(\text{senderEntityModel}) \wedge$ $(\forall \text{senderInteractorModel:INTERACTOR_MODELS};$ $\text{message : MESSAGES};$ $\text{recipientInteractorModel:INTERACTOR_MODELS} \bullet$ $\text{senderInteractorModel} \in \text{ent_model_registered_interactors}(\text{senderEntityModel}) \wedge$ $\text{message} = \text{incoming_queue}(\text{intr_model_id}(\text{intr_model_metadata}(\text{senderInteractorModel}))) \wedge$ $\text{recipientInteractorModel} = \text{bound}(\text{senderInteractorModel}) \wedge$ $\text{outgoing_queue} = \text{outgoing_queue} \cup$ $\{\text{intr_model_id}(\text{intr_model_metadata}(\text{recipientInteractorModel})) \mapsto \text{message}\})$
--

Listing 4-23. Message mediation within a UA.

4.2. Specifying the Middleware API

In this section we specify the middleware API (Application Programming Interface) that enables DIS (Distributed Interactor Software) to connect to the middleware and participate in UA sessions. We specify an API providing the main elements for the interaction between the DIS and the middleware allowing for the establishment of service provider-consumer relationships and communications as required by the proposed SOA framework. The API transparently provides to the DIS the following functions:

- “apiRegistration” in Listing 4-29 for registering with the middleware. Physical sensors and actuators register as service providers by sending their descriptions to the middleware to advertise their services and make them discoverable. On the other hand the agent sensors and actuators register as service consumers by sending the descriptions of the services that they require in order for the middleware to bind them to the providers of these services.

- “apiGetMessageMetadata” in Listing 4-30. This API function uses the service description (that was sent to the middleware by the “apiRegistration”) to create new metadata for describing sensing/acting requests or action feedback/sensory data messages. It returns this metadata to the DIS.
- “apiEditMessageMetadata” in Listing 4-32. The DIS uses the message metadata that it has acquired from the “apiGetMessageMetadata” function and edits it using this function. In particular it adds at runtime parameterized data values describing sensing/acting requests or action feedback/sensory data messages. This way it creates messages using structured metadata that contains values that describe particular events.
- “apiSendMessage” in Listing 4-33. It is used by the DIS for sending the messages that were created using “apiEditMessageMetadata” to a DIS that is bound to it via the middleware.
- “apiReceiveMessage” in Listing 4-34 for receiving messages from a DIS that is bound to it via the middleware.
- “apiProcessMessage” in Listing 4-35. Specifies the API operation for enabling the DIS to extract the values that describe a sensing/action request or sensory data/feedback from message metadata that is acquired from a received message (see “apiReceiveMessage” operation).

The framework for the middleware API was also type-checked using the fuzz type-checker [48].

4.2.1. API Definition

We specify the main sets for the API framework as follows:

```
[INTERACTOR_IDENTIFIERS, ENTITY_IDENTIFIERS, SERVICE_DESCRIPTIONS,  
TARGET, MESSAGES, METADATA, DIS_INPUT_DATA]
```

Most of the names here are self-explanatory and the descriptions of the sets can be found in appendix A-2. We use the above sets to create variables that will constitute the data that is stored by the API in order to implement its functionalities. Now we can define the schema for the API framework:

```
API  
-----  
descriptionMetaData: METADATA  
ID : INTERACTOR_IDENTIFIERS  
type : INTERACTOR_TYPE  
binding_type: BINDING_TYPE;  
locTarget: TARGET  
entity_ID: ENTITY_IDENTIFIERS;  
entity_type: ENTITY_TYPE  
body_ids:  $\mathbb{P}$  INTERACTOR_IDENTIFIERS  
service_type : SERVICE_TYPE  
service_description : SERVICE_DESCRIPTIONS  
-----
```

Listing 4-24. The schema of the middleware API.

In particular, the “descriptionMetaData” variable specifies metadata describing the interactor by association with the rest of the sets/variables of Listing 4-24. In the following we will see how we create these associations (Listing 4-27). We also define free types which are sets containing exact numbers of predefined values (their meaning is also described in appendix A-2):

```
BINDING_TYPE ::= targeted |agnostic  
INTERACTOR_TYPE ::= sensor |actuator  
ENTITY_TYPE ::= agent |avatar  
SERVICE_TYPE ::= provide |consumer
```

We use the API schema to define the “api” variable allowing us to access the elements of the schema globally in order to help us specify the functionality of the API.

```
| api:API
```

Listing 4-25. An “API” object for accessing the variables of the API schema.

The middleware supports two types of messages that contain: metadata describing interactors for registration purposes and metadata that describes control/sensing requests and action feedback/sensory data. We specify a support function for creating such middleware messages using metadata:

```
| metadataToMessage: MESSAGES → METADATA
```

Listing 4-26. Adding metadata to a middleware message.

We note that the operations of the specification that will be prefixed with “api” specify the middleware API functions that are accessible by the DIS.

4.2.2. Registration

The system designer describes an interactor with metadata that is stored on the hard disk. The API uses the following functions to support the functionality for reading and storing internally this metadata. The functions of Listing 4-27 are described in appendix A-3.

```
| idToMetaData: METADATA → INTERACTOR_IDENTIFIERS  
| typeToMetaData: METADATA → INTERACTOR_TYPE  
| bindingTypeToMetaData: METADATA → BINDING_TYPE  
| targetToMetaData: METADATA → TARGET  
| entityIdToMetaData: METADATA → ENTITY_IDENTIFIERS  
| entityTypeToMetaData: METADATA → ENTITY_TYPE  
| bodyIdsToMetaData: METADATA → PINTERACTOR_IDENTIFIERS  
| serviceTypeToMetaData: METADATA → SERVICE_TYPE  
| serviceDescriptionToMetaData: METADATA → SERVICE_DESCRIPTIONS
```

Listing 4-27. API functions for acquiring metadata that describes the interactor.

We are particularly interested in the service description that is part of the metadata and can be retrieved using the “serviceDescriptionToMetaData” function. The service description is created by the system designer and it is a description of interactor sensing or acting functionality as service providing or service consuming based on the service type. At this point we can define the function that reads the entire user defined data (e.g. stored in the hard disk).

$\text{readMetaData} : (\text{INTERACTOR_IDENTIFIERS} \times \text{INTERACTOR_TYPE} \times \text{BINDING_TYPE} \times \text{TARGET} \times \text{ENTITY_IDENTIFIERS} \times \text{SERVICE_TYPE} \times \text{SERVICE_DESCRIPTIONS} \times \text{ENTITY_TYPE} \times \mathbb{P} \text{ INTERACTOR_IDENTIFIERS}) \rightarrow \text{METADATA}$
$\exists \text{metadata: METADATA} \bullet$ $\text{readMetaData} (\text{api.ID}, \text{api.type}, \text{api.binding_type}, \text{api.locTarget}, \text{api.entity_ID}, \text{api.service_type}, \text{api.service_description}, \text{api.entity_type}, \text{api.body_ids}) = \text{metadata} \Leftrightarrow$ $\text{idToMetaData} = \text{idToMetaData} \cup \{ \text{metadata} \mapsto \text{api.ID} \}$ $\wedge \text{typeToMetaData} = \text{typeToMetaData} \cup \{ \text{metadata} \mapsto \text{api.type} \}$ $\wedge \text{bindingTypeToMetaData} = \text{bindingTypeToMetaData} \cup \{ \text{metadata} \mapsto \text{api.binding_type} \}$ $\wedge \text{targetToMetaData} = \text{targetToMetaData} \cup \{ \text{metadata} \mapsto \text{api.locTarget} \}$ $\wedge \text{entityIdToMetaData} = \text{entityIdToMetaData} \cup \{ \text{metadata} \mapsto \text{api.entity_ID} \}$ $\wedge \text{entityTypeToMetaData} = \text{entityTypeToMetaData} \cup \{ \text{metadata} \mapsto \text{api.entity_type} \}$ $\wedge \text{bodyIdsToMetaData} = \text{bodyIdsToMetaData} \cup \{ \text{metadata} \mapsto \text{api.body_ids} \}$ $\wedge \text{serviceTypeToMetaData} = \text{serviceTypeToMetaData} \cup \{ \text{metadata} \mapsto \text{api.service_type} \}$ $\wedge \text{serviceDescriptionToMetaData} = \text{serviceDescriptionToMetaData} \cup \{ \text{metadata} \mapsto \text{api.service_description} \}$

Listing 4-28. Getting the metadata that describes the interactor.

Now we can proceed with specifying the registration operation that is available to the DIS. For this operation, the interactor metadata description is acquired using the “readMetaData” function and stored in “api.descriptionMetaData”. We remind that “api.descriptionMetaData” contains information (“SERVICE_TYPE” value) describing the interactor as a provider of a service or as a consumer of a service that is provided by other interactors. Then the “api.descriptionMetaData” is added to a registration message that will be sent to the middleware. The operation for the registration is specified with the following schema:

apiRegistration <hr/> $\text{registrationMessage!}: \text{MESSAGES}$ <hr/> $\text{api.descriptionMetaData} = \text{readMetaData} (\text{api.ID}, \text{api.type}, \text{api.binding_type},$ $\text{api.locTarget}, \text{api.entity_ID}, \text{api.service_type}, \text{api.service_description} ,$ $\text{api.entity_type}, \text{api.body_ids}) \wedge$ $\text{metadataToMessage} = \text{metadataToMessage} \cup$ $\{\text{registrationMessage!} \mapsto \text{api.descriptionMetaData}\}$
--

Listing 4-29. The registration of the DIS.

We are not interested in the low-level communication protocol that ensures that the message is received by the middleware e.g. by the latter sending back a confirmation as how this is achieved is usually a matter of implementation. Going into such detail in the specification of the framework could limit the adaptability of the framework in terms of application in a variety of settings.

4.2.3. Messaging

The API provides the DIS with the metadata that will be used for the creation of messages describing sensing and acting requests by agent sensors and actuators and sensory data or action feedback by the physical sensors and actuators. The metadata it uses includes the identifier of the interactor (“idToMetaData” function), the identifier of the entity (“entityIdToMetaData” function) and the service description.

$\text{apiGetMessageMetadata}: \text{METADATA} \longrightarrow \text{METADATA}$ <hr/> $\exists \text{messageMetaData}: \text{METADATA} \bullet$ $\text{apiGetMessageMetadata}(\text{api.descriptionMetaData}) = \text{messageMetaData} \Leftrightarrow$ $\text{idToMetaData} = \text{idToMetaData} \cup$ $\{\text{messageMetaData} \mapsto \text{idToMetaData} (\text{api.descriptionMetaData}) \}$ $\wedge \text{entityIdToMetaData} = \text{entityIdToMetaData} \cup$ $\{\text{messageMetaData} \mapsto \text{entityIdToMetaData} (\text{api.descriptionMetaData}) \}$ $\wedge \text{serviceDescriptionToMetaData} = \text{serviceDescriptionToMetaData} \cup$ $\{\text{messageMetaData} \mapsto \text{serviceDescriptionToMetaData}(\text{api.descriptionMetaData}) \}$
--

Listing 4-30. This API function uses the service description element of the metadata description to create messaging metadata for describing sensing/acting requests or action feedback/sensory data messages.

The DIS use textual or numerical values to parameterize the aforementioned sensing/acting requests as well as the sensory data and action feedback. Then, they edit the metadata that they have acquired from the “apiGetMessageMetadata” function by adding these values. For example, in order for an actuator to issue a “switch the light on” request, it will need to use the metadata that describes the switching of the light and add the value “ON” to it. We specify the API support function for attaching the data input to the metadata that will be send as part of the middleware message:

$$\left| \text{propertiesToMetadata: METADATA} \rightarrow \text{DIS_INPUT_DATA} \right.$$

Listing 4-31. Adding input data values describing interactor activity to the metadata of a sensing/acting message.

The API function that enables the DIS to create a message using the message metadata from the “apiGetMessageMetadata” and the parameterized data input is specified as follows:

$$\left| \begin{array}{l} \text{apiEditMessageMetadata: METADATA} \times \text{DIS_INPUT_DATA} \rightarrow \text{METADATA} \\ \hline \exists \text{messageMetaData: METADATA;} \\ \text{messageToSend: METADATA;} \\ \text{dis_input_data: DIS_INPUT_DATA} \bullet \\ \text{apiEditMessageMetadata (messageMetaData, dis_input_data)} \\ = \text{messageToSend} \Leftrightarrow \\ \text{propertiesToMetadata} = \text{propertiesToMetadata} \cup \{ \text{messageMetaData} \mapsto \text{dis_input_data} \} \\ \wedge \text{messageToSend} = \text{messageMetaData} \end{array} \right.$$

Listing 4-32. The DIS uses the message metadata that it has acquired from the “apiGetMessageMetadata” function and adds to it at runtime the parameterized data values describing a sensing/acting request or an action feedback/sensory data message.

The “apiSendMessage” function then specifies how the DIS uses the metadata that was created by “apiEditMessageMetadata” to create the message during runtime that will be sent to the middleware:

apiSendMessage messageMetaData?: METADATA messageToSend!: MESSAGES
metadataToMessage = metadataToMessage ∪ { messageToSend! ↦ apiEditMessageMetadata (messageMetaData?, dis_input_data?) }

Listing 4-33. Operation that is used by DIS for sending messages to the middleware. The DIS creates the messages using the “apiEditMessageMetadata” function.

Finally we specify the operation for receiving a message from the middleware. The DIS will use the following operation to receive messages from the middleware. The operation acquires the metadata from the received message using the “metadataToMessage” function.

apiReceiveMessage message?: MESSAGES messageMetaData!: METADATA
messageMetaData! = metadataToMessage (message?)

Listing 4-34. The operation for receiving a message from the middleware and returns the metadata of the message.

The next operation enables the DIS to use the metadata of the received message to extract data that is similar to the data input of the “apiSendMessage” operation. This data will then be passed to the DIS (“properties!” variable) in a format that allows it to process it. For example when a light switch actuator receives a message that contains metadata with the value “ON” it will acquire this value and act accordingly.

apiProcessMessage messageMetaData!: METADATA properties!: DIS_INPUT_DATA
properties! = propertiesToMetadata(messageMetaData!)

Listing 4-35. Specifies the API operation for enabling the DIS to extract the values that describe a sensing/action request or sensory data/feedback. The message metadata is acquired from a received message using the “apiReceiveMessage” operation.

4.3. Summary

The main contribution of this chapter is a framework that uses the Z-Notation to specify a middleware for creating UAs. The framework is based on the requirements of chapter 3 and its purpose is to guide the development of such middleware. Having built a basic framework, we can extend it by adding monitoring functionality and more useful functions that reason upon the internal representation objects within the middleware. The middleware framework will be used, as we will see in the following chapter, for the design and implementation of the eVATAR middleware.

5. EVATAR

In this chapter we present the eVATAR middleware. The aim is to show how we can use the architecture and framework presented in chapters 3 and 4 to implement a middleware for UAs. The main contribution of this chapter is to describe how eVATAR implements a message oriented middleware as illustrated in the architecture of Figure 3-6 with a second layer of reflective functionality. This chapter presents the current state of the implementation of eVATAR and does not discuss improvements or future work (see chapter 8).

We first provide the reader with a description of the base layer of eVATAR and its main components: the connection server, the interactor registration and the message queue components. We will also describe the BIL (Body Integration Language) metadata language. BIL describes physical and agent interactors as service providers and consumers and also provides with protocols for their communication with eVATAR. We then describe the eVATAR API that enables distributed interactor software to interact with the base layer of eVATAR. A description of the reflection layer of eVATAR follows that implements the “binding”, “models”, “ubiquitous agents” and “monitoring” components (see architecture in 3.4).

5.1. The Base Layer of eVATAR

eVATAR implements the base layer functionality of our middleware architecture following service oriented principles ([57], [58] and [89]). The agent interactors register to eVATAR as service consumers while the avatar interactors register as service providers. eVATAR and the eVATAR API implement the communication protocols between the service providers and the service consumers. eVATAR uses BIL (Body Integration Language) that provides ontology for the communication. Before we proceed with the description of the base layer of eVATAR it would be useful to define and present BIL.

5.1.1. The BIL Metadata Language

BIL is an XML [107] based metadata language. BIL types constitute the foundation for the XML schemas against which all BIL documents may be evaluated. XML allows describing the types of documents using XML Schema Definitions (XSD). Such definitions specify the document structure with a list of legal elements and attributes [107]. The full XSDs for BIL are provided in Appendix B-1. BIL uses two schemas; the first is used for creating BIL descriptions of interactors (sensors/actuators) and the second for creating BIL messages useful for sending and receiving commands, sensory data and feedback. The following diagram illustrates the BIL description:

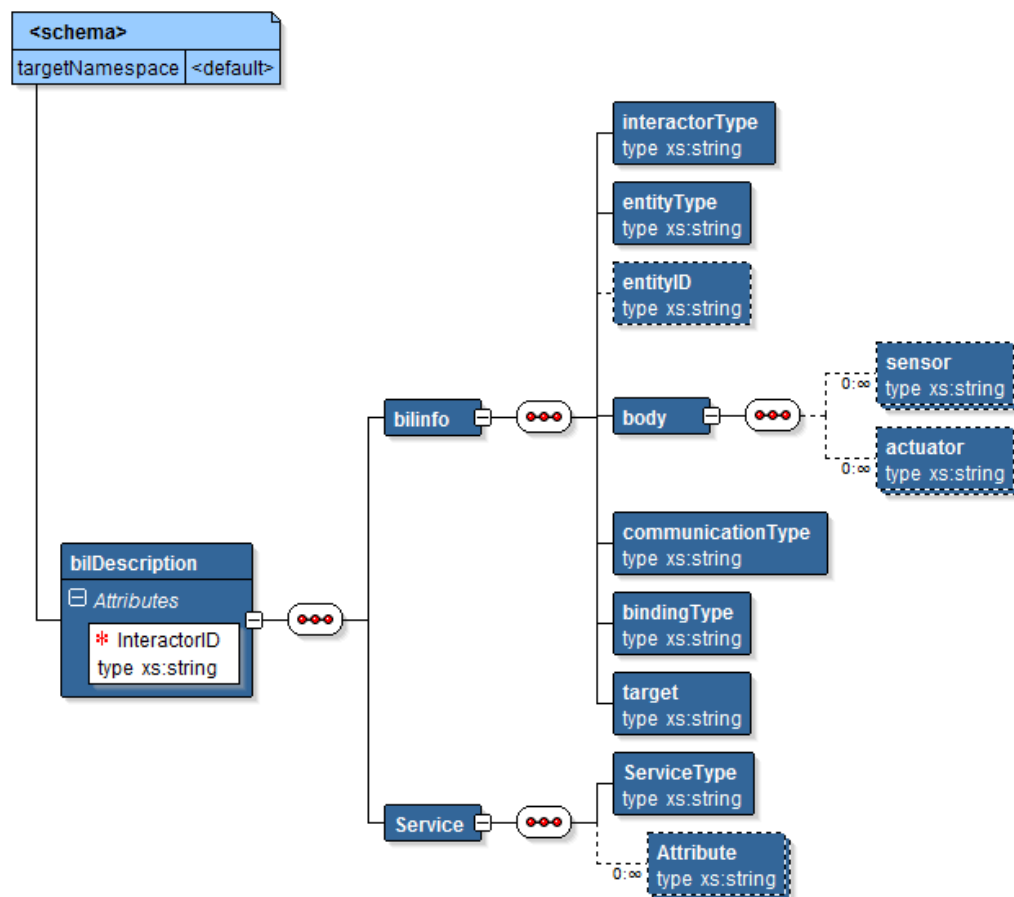


Diagram 5-1. BIL description representation based on the bilDescription.xsd in Appendix B-1.

The BIL description contains the metadata elements specified in the “BS_IncomingRegistrationData” specification of Listing 4-3 in the Z-Notation

framework of chapter 4. The root element of the interactor BIL description is named "bilDescription" and it contains a required attribute called "InteractorID" that takes as a value the unique identifier of the BIL description for the interactor. The root element has two child elements, the "bilinfo" and the "Service". The following table describes all the child elements of "bilinfo":

Element	Description
interactorType	One of: "SENSOR" and "ACTUATOR".
entityType	One of: "AGENT" and "AVATAR".
entityId	The id (string value) of the entity that this interactor belongs to.
body	Contains two child elements, the "sensor" and the "actuator". Both of these elements take sensor and actuator identifier values in the form of strings. They comprise a set including all the interactor identifiers that are associated with a particular agent or avatar (whose identifier is the string of the entityID tag).
communicationType	It defines the message exchange protocol between the service provider and the consumer at application level. Currently supported value: "REQUESTREPLY".
bindingType	It is used to instruct the middleware what type of binding should be pursued, "targeted" or "agnostic" (see 3.4)
Target	If the binding type is "targeted", then the middleware is instructed to only allow bindings between interactors that have BIL descriptions sharing the same value in their "target" tags. The value is "na" (not applicable) for agnostic binding.

Table 5-1. The child elements of the "bilinfo" element that describe an interactor.

Below we can see an example of a BIL description of an agent actuator based on the first schema.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bilDescription InteractorID ="actuator2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bildescription.xsd">
<bilinfo>
<interactorType>ACTUATOR</interactorType>
<entityType>AGENT</entityType>
<entityID>agent1</entityID>
  <body>
    <sensor>sensor1</sensor>
    <sensor>sensor2</sensor>
    <sensor>sensor3</sensor>
    <actuator>actuator1</actuator>
    <actuator>actuator2</actuator>
  </body>
<CommunicationType>REQUESTREPLY</CommunicationType>
<bindingType>agnostic</bindingType>
<target>na</target>
</bilinfo>
<Service>
<ServiceType>CONSUMER</ServiceType>
<Attribute>StatusOn</Attribute>
<Attribute>ValueInt</Attribute>
</Service>
</bilDescription>
```

Figure 5-1. BIL description of an actuator.

The snippet of Figure 5-1 which is created based on the BIL description XSD that can be found in Appendix B-1 indicates that a “body” element may contain from none to as many “sensor” and “actuator” elements as the author of the BIL XML description wishes.

```
<xs:element name="sensor" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element name="actuator" type="xs:string" minOccurs="0"
maxOccurs="unbounded">
```

Figure 5-2. BIL snippet from the BIL description schema of Appendix B-1.

The above set of interactor identifiers in the BIL description of an interactor is used by eVATAR to determine when an agent or an avatar (with the value of the entityID as identifier) is fully registered.

The second child element of the root that is tagged as “Service” contains two elements as we can see in the following table:

Element	Description
ServiceType	It denotes whether the interactor is a provider or a consumer of a service, accepting the following strings: “PROVIDER” and “CONSUMER”.
Attribute	This element takes a string that could have the following values:{"ValueInt", "ValueLong", "TEXT", "XML", "StatusOn", "Speed", "Temperature", "Duration", "Height", "Length", "ASSERT"}.

Table 5-2. The child elements of the “Service” element that describe a service.

It is up to the author of the XML document to decide how many “Attribute” child elements of the “Service” element the document will have. Each of them will be bearing a value from the set of strings in Table 5-2. It needs to be noted that Table 5-2 only includes a representative set of string inputs for the “Attribute” element and the eVATAR implementation uses more. These values help describe a service whether it is the service offered by the particular interactor or the service that will be consumed by the particular interactor if it is a consumer. They will be used by eVATAR to find a compatible for connection interactor (see compatibility evaluation in 3.3.3 and 4.1.7). All interactor BIL descriptions can be validated against the BIL interactor schema in Appendix B-1.

The second schema in BIL defines the messages between the interactors (see second XSD BIL message in Appendix B-1). The diagram for the particular XSD:

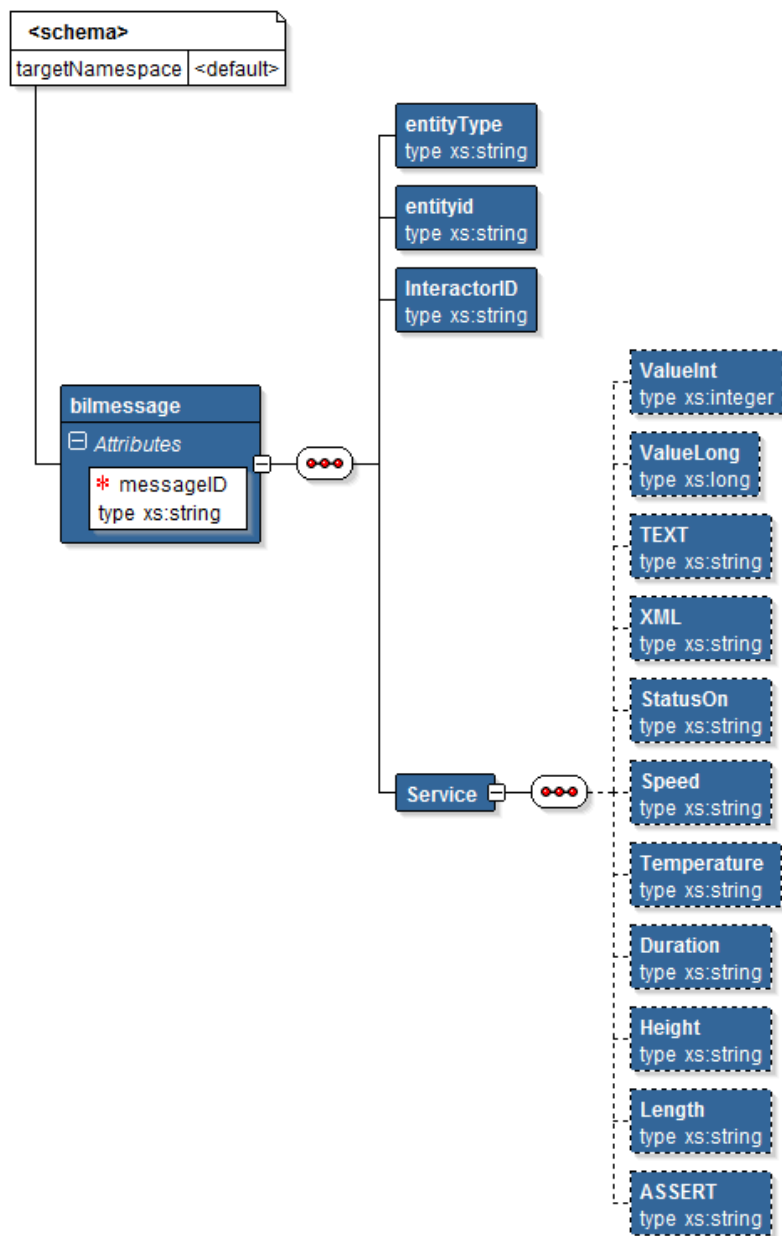


Diagram 5-2. BIL message representation based on the bilMessage.xsd in Appendix B-1.

An example of a BIL message that can be validated against the BIL message schema in Appendix B-1:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<bilmessage messageID="message2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bilmessage.xsd">
<entityType>AGENT</entityType>
<entityid>agent1</entityid>
<InteractorID>actuator2</InteractorID>
<Service>
<StatusOn>ON</StatusOn>
<ValueInt>1</ValueInt>
</Service>
</bilmessage>

```

Figure 5-3. BIL Message.

The “SenderInteractorID” element accepts a string value with the sender interactor’s identifier. All of the child elements of “Service” are optional (see bilMessage XSD in Appendix B-1). For example:

```

<xs:element name="ValueInt" type="xs:integer" minOccurs="0"/>
<xs:element name="XML" type="xs:string" minOccurs="0"/>
<xs:element name="StatusOn" type="xs:string" minOccurs="0"/>

```

Figure 5-4. Example of child elements in the “Service” tag.

In Figure 5-4 we see that all elements state that `minOccurs="0"`. The above list is representative, similarly to the list of values for the “Attribute” child elements of the “Service” element in the BIL interactor description. Table B-0-1 in Appendix B describes the child elements of the “Service” tag for the BIL message explaining their uses. In all elements of Table B-0-1, if the value is “GET” instead of the defined acceptable character strings, it means that this is a sensing request. For example a “<Temperature>GET</Temperature>” tag would belong to a temperature sensing request from an agent sensor.

BIL interactor descriptions register all the service information of the interactor with eVATAR as a combination of “Attribute” children of the “Service” element and BIL messages use the values of these “Attribute” elements as the children of their own “Service” element. For example in the BIL description of the actuator we saw in Figure 5-1:

```
<Attribute>StatusOn</Attribute>
```

In the BIL message of Figure 5-3 we saw the corresponding element as:

```
<StatusOn>ON</StatusOn>
```

BIL is used instead of using an existing service definition language such as WSDL (Web Services Description Language) [132]. BIL is light weight for domain specific descriptions needed for eVATAR interactions. BIL descriptions are similar in concept and in principle to the ones of the WSDL that is used to describe web-services. WSDL is a well-known and mature technology. BIL on the other hand is a purpose built language for the definition of agent/avatar bodies leading to the creation of compact service descriptions. In other words, BIL aspires to a reduced message processing overhead and network traffic for a specific application as opposed to WSDL which is designed for different types of applications and web services.

5.1.2. Base Layer Components

eVATAR is implemented in Java. The base layer of eVATAR is responsible for the MOM functionality of eVATAR. Following the architecture that was described in 3.4, the base layer should implement the connection server, the interactor registration and the message queue components. In Diagram 5-3 below, we see a class diagram for the implementation of the base layer JAVA package. The diagram shows class names and relations between them while the more detailed diagram for every class can be found in Appendix B-3, in section “eVATAR Base Layer UML classes”. The main class of this component is called “evatarMain”.

The “evatarMainProcess” function (see Appendix B-3, Diagram B-1) implements the main thread of execution of the middleware. All other threads are children of the “evatarMainProcess” thread. It is responsible for running the initialization function (from the “initialization” class) at the start-up of the

system in order to initialize all important data structures by providing them with their startup values in both base and reflection layers of eVATAR.

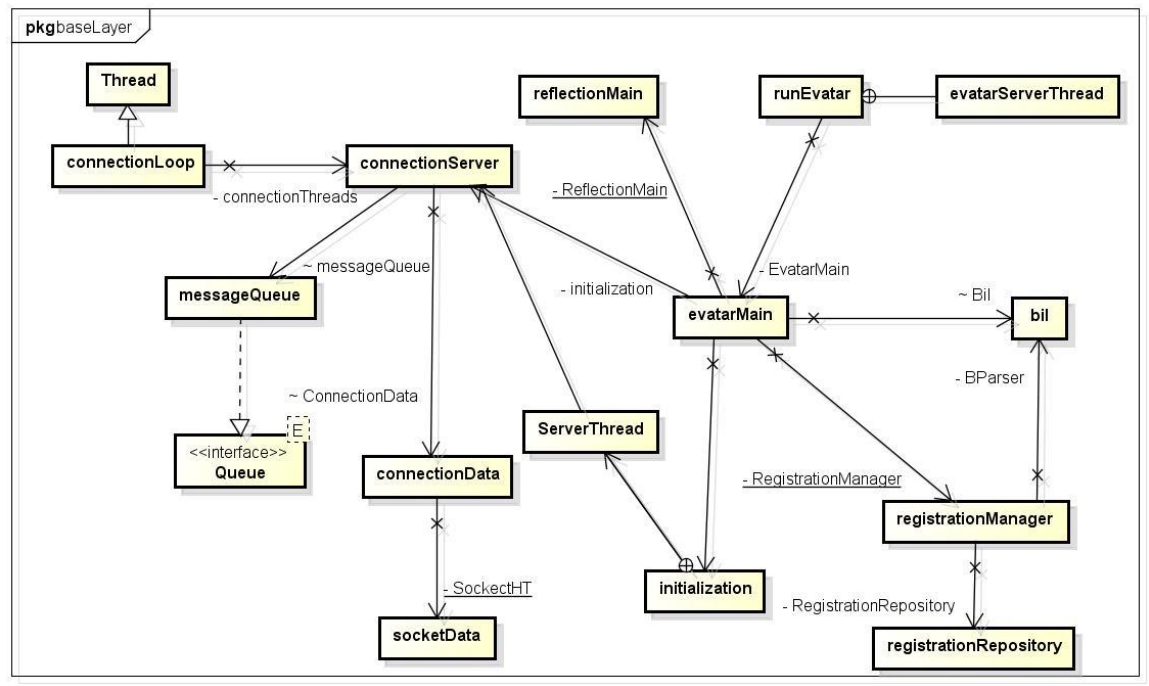


Diagram 5-3. Base layer class diagram (for UML class details see Appendix B-3).

The “evatarMainProcess” also creates the connection server object (implementing the “connectionServer” class) that will be running a separate thread of program execution. The connection server manages the connectivity between eVATAR and the distributed interactor software that connects to it as service providers and consumers. The connection server creates a TCP [49] server for accepting connections. It is a multithreaded server creating a new, separate thread of execution using the “connectionLoop” class (see Diagram B-4 in Appendix B-3) to serve every new connection. All connections use blocking TCP sockets meaning that they will not return control to the thread of execution they are called from until a message has been sent or received [49]. The connection server also updates the “connectionData” storage facility that implements JAVA hash tables [103] that store information about the TCP connections. The following sample program code illustrates the main logic behind the implementation of the “connectionLoop” thread.

```

while (RUNNING) {
    // send messages from the outgoing message queue to the
    // interactor using the TCP connection
    outgoingQueue.send();

    // read the next message in the incoming message queue
    message = incomingQueue.nextMessage();

    // if the message is not null and it is a BIL description
    if (isDescription(message) == true) {

        // update the registry and . . .
        registrationManager.update(message);

        // notify the reflection layer that a new interactor
        // is registered
        reflectionMain.notification(interactorID);
    }
    else {
        // do nothing
        // the messages will be processed by the reflection layer
    }
}
}

```

Figure 5-5. Sample code describing the functionality of a “connectionLoop” object.

All incoming messages via the connections’ TCP socket are placed in a FIFO (first in first out) message queue while messages from eVATAR to the interactors are placed in an identical outgoing FIFO queue. The queues are implemented using the “messageQueue” class which implements the “Queue” JAVA interface [110]. The queues support multi-threaded access (from the connectionLoop thread and from threads in the reflection layer of eVATAR as we will see in the following).

The “connectionLoop” object sends messages in outgoing queues to the interactors using the TCP connection while it also reads the incoming queue for messages containing BIL descriptions. BIL messages in the incoming queue that do not contain descriptions are processed directly by the reflection layer (see section 5.2.2). The “bil” class offers eVATAR the functionality to process BIL descriptions and messages (see Diagram B-9 in Appendix B-3).

If the message is a BIL description the “connectionLoop” object will notify the reflection layer functionality that a new interactor has been registered and it will also forward the description to the service registration functionality of the

base layer of eVATAR. The “service registration” component of the architecture that was described in 3.4 is implemented by two classes in Diagram 5-3: the “registrationRepository” and the “registrationManager”. Messages containing BIL descriptions of agent or avatar interactors will update the registration repository¹.

The “registrationManager” class manages the way interactors register with the system (and in particular with the repository). It implements the functionality that is described in sections 4.1.3 and 4.1.6 of the Z-Notation framework². The data stored in the registration repository is used for the creation of the entity/interactor models in the reflection layer as we will see in the following.

The “evatarMain” class also features a service continuity mechanism. At startup, it seeks for a file in a predetermined, by eVATARS’ configuration, location in the hard disk. We call this the “heartbeat” file. If the file does not exist or it is not updated within a predetermined time interval, eVATAR will assume that no other instance of eVATAR is running. It will then perform a system call to create a new eVATAR process and proceed with its middleware tasks. The “evatarMainProcess” will be updating the heartbeat file regularly. The second eVATAR process will detect the existence of a heartbeat file and it will then assume the role of the monitor.

In the monitor eVATAR, the “evatarMainProcess” will be running the “monitoringLoop” function (see Diagram B-1 in appendix B-3) that loops in predetermined timed intervals checking whether the heartbeat file has been updated. In the case of failure of the active eVATAR, the monitor eVATAR will identify the failure (heartbeat reading failure) and assume the role of the active middleware. We call this mechanism as the “failover”. It will be using the same network address allowing the distributed interactor software to reconnect

¹ It implements the “registration_list” of Listing 4-4 in the Z-Notation specification of 4 using hash tables that store BIL metadata. See Diagram 5-3.

² As we can see in Appendix B-3 it implements the registration functions of Listing 4-4 in the framework as well as the functionality for acquiring a BIL description from the connection server, processing it and storing it to the registration repository as specified in the “BS_ReceiveAndProcessRegistrationData” and “RegisterInteractor” operations in Listing 4-5 and Listing 4-15.

aiming to maintain service continuity. The interactors will have to register again with the new instance of eVATAR. Finally, the “evatarMain” creates the “reflectionMain” object which is the main class of the reflection layer that manages the reflective functionality. It provides functionality in a separate thread of program execution to the one of the “evatarMainProcess”. We will describe the “reflectionMain” class in more detail in section 5.2.2.

5.1.3. eVATAR API

The eVATAR API enables DIS (Distributed Interactor Software) to register with eVATAR as service providers or service consumers and to also exchange messages with other DIS via eVATAR. The functionality of the eVATAR API was specified in section 4.2 of chapter 4. The listings that will be referenced in this section are also specified in 4.2. The prototype of the eVATAR API is a JAVA jar [113] file and this section will present the JAVA classes that implement its functionality. The class diagram for the eVATAR API:

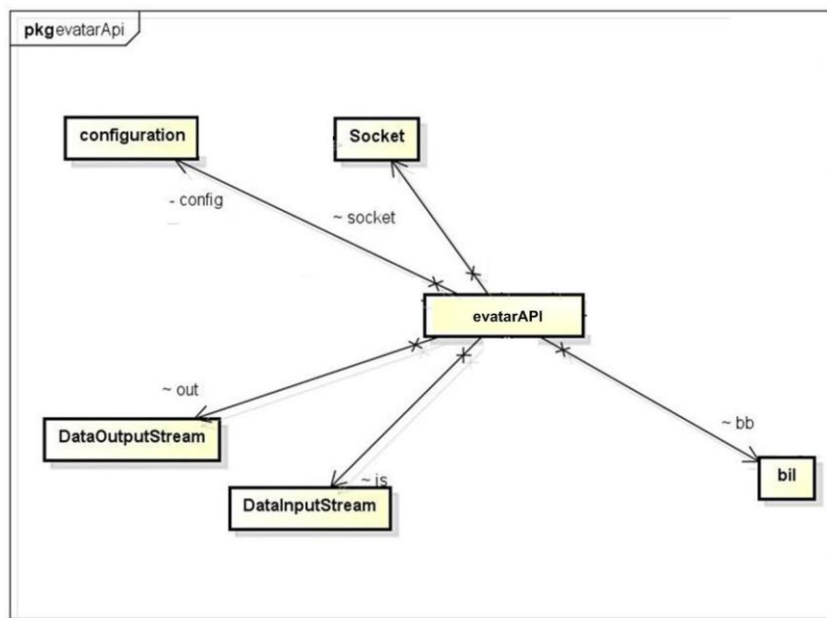


Diagram 5-4. The eVATAR API class diagram (class details in Appendix B-3).

The system developer defines the XML based metadata description of the functionality that the DIS provides or consumes and in general all the

information required for the DIS to participate in the system by creating a BIL description document and storing it in the hard disk for the API to access.

The “evatarAPI” class (Diagram 5-4 and Appendix B-3) provides functionality for starting a TCP [49] client that connects to the eVATAR server and enables the exchange of messages. The main API functions that are available to the DIS are prefixed with the word “api” and can be found in Diagram B-11. The “evatarAPI” class provides the DIS with the API functions that were described in section 4.2 of chapter 4.

In particular it enables DIS to send a BIL description to eVATAR for registration purposes. This way it implements the “apiRegistration” operation of Listing 4-29. The “bil” class (see Diagram B-12 in Appendix B-3) enables the API to process BIL metadata. It features internal data structures for storing information that describes the interactor (Listing 4-27). When the DIS initialise the eVATAR API, the latter uses “bil” to acquire the BIL description that is stored on the hard disk by the system designer and to update the relevant “bil” data structures (Listing 4-28). The “bil” class also creates BIL message metadata based on the BIL message XSD (see Appendix B-1). It returns this metadata to the DIS (see the “apiGetBilMessageMetadata” function in Diagram B-11 that implements the specification of Listing 4-30).

The API also returns a “bil” object to the DIS via the “apiGetBilObject” function (Diagram B-11) enabling it with the capability to also process BIL metadata. It uses the acquired BIL description metadata and in particular the service element (Table 5-2) to create messages that describe action/sensing requests, action feedback and sensory data messages. During the runtime, the DIS uses the functions that are available via the “bil” object to edit the BIL message and add to it the relevant values that describe a sensing or an acting event or request (implementing Listing 4-32). The API provides via the “bil” class with an extensive list of functions for reading and updating the important tags and attributes of BIL items (see Diagram B-12 in Appendix B-3). It offers “set” and “get” functions for every attribute. In the example of Figure 5-3 this would entail adding the appropriate values to the following tags:

```
<StatusOn>ON</StatusOn> % adding the "ON" value  
<ValueInt>1</ValueInt> % adding the value 1
```

Furthermore, the API provides functions for sending BIL messages to eVATAR (implementing the "apiSendMessage" function in Listing 4-33) or receiving BIL messages from it ("apiReceiveMessage" in Listing 4-34). With regards to the received messages, the "bil" object allows the recipient DIS to extract the values from the BIL message (in the above example the extracted values would be "ON" for the "statusOn" tag and "1" for the "ValueInt" tag).

5.2. The Reflection-layer of eVATAR

The reflection layer implements the models, binding, ubiquitous agents and monitoring components of the middleware architecture that was depicted in Figure 3-6. The implementation uses the JAVA language. There are two JAVA packages implementing the functionality of the reflection layer: the "models" and the "reflection" package.

5.2.1. Models

The following UML diagram describes the "models" package:

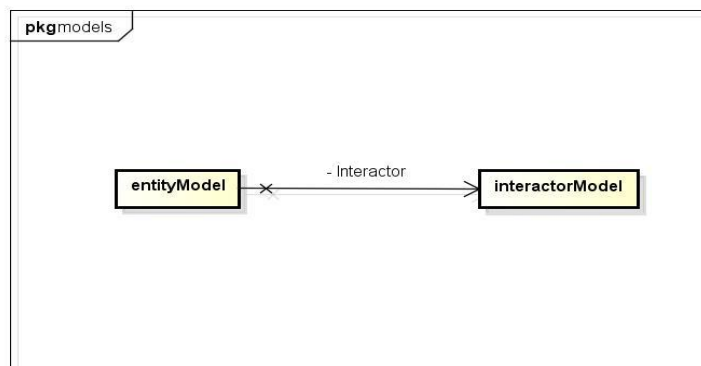


Diagram 5-5. The "models" package class diagram (for UML class details see Appendix B-3).

The entity and interactor models that were described in sections 3.3.2 and 3.4 are implemented as JAVA classes. An entity model object (from the “entityModel” class) contains information about the entity and the interactors that constitute its body (3.3.2). Following the specification of Listing 4-6 in chapter 4, the metadata of the “entityModel” class contains:

- The identifier of the entity as a string of characters.
- The type of the entity (character string “agent” or “avatar”).
- A list containing the identifiers of the interactors that belong to the particular entity and have already been registered with eVATAR.
- A list containing the identifiers of all the interactors that are expected to register with eVATAR and belong to the body of the particular entity.

It also implements functions for accessing and updating the above variables and data structures. Furthermore, the interactor models can be accessed via the “entityModel” class. The latter features a function that accepts the interactor identifier as input and if it belongs to the list of registered identifiers for the particular entity the function will return the corresponding interactor model. The class diagram of the “entityModel” class can be found in Diagram B-14 of Appendix B-3.

The interactor models reflect to the base interactors of agent or avatar bodies. They are also implemented in Java. Objects implementing the “interactorModel” class consist of two layers called the data layer and the interactivity layer. Similarly to the entity model, the data layer features metadata describing the base agent/avatar interactor. As per Listing 4-11, it contains the following metadata:

- The identifier of the interactor as a string of characters.
- The identifier of the entity it belongs to as a string of characters.
- The type of the interactor (character string “sensor” or “actuator”).

- The service type as a character string for service: “provider” or “consumer”.
- The binding type as a character string with values: “targeted” or “agnostic”.
- A target value as an integer value.
- A service description as a string containing a description of the service that the interactor provides or consumes that is described using XML syntax.

The interactor model provides with functions for updating and querying the above variables and data structures. The interactivity layer of the interactor is implemented as two message queues, one for incoming and one for outgoing messages (*Listing 4-13*). The queues are the same queues that are used in the connection server (base layer) for the communication with the distributed interactor software. We remind that these queues allow multi-threaded access. At this stage in the UA lifecycle, these queues contain only BIL messages for control or sensing and not descriptions. This way we implement a direct link between an interactor model and the TCP connection in the base layer that allows it to send and receive messages to the distributed interactor that it models in order to implement reflection (see 2.1 in the “Background” chapter). The “interactorModel” class uses the “createEvent()” and “receiveEvent()” functions to update the outgoing and incoming queues respectively. The first one models a message from the interactor (for an action or a sensing) while the second models a message towards the particular interactor. The Diagram 5-5 shows class names for the “models” package and the relations between them while the more detailed diagrams for every class can be found in Appendix B-3.

5.2.2. Reflection

The “reflection” package implements the functionality for creating and updating with new information the entity and interactor models. It also

performs reflective binding and creates UAs following the requirements of chapter 3 and the specification of chapter 4. The Diagram 5-6 shows the names of the classes that implement the “reflection” package and the relations between them while the more detailed diagram of every class can be found in Appendix B-3, in section “Reflection functionality package UML classes”.

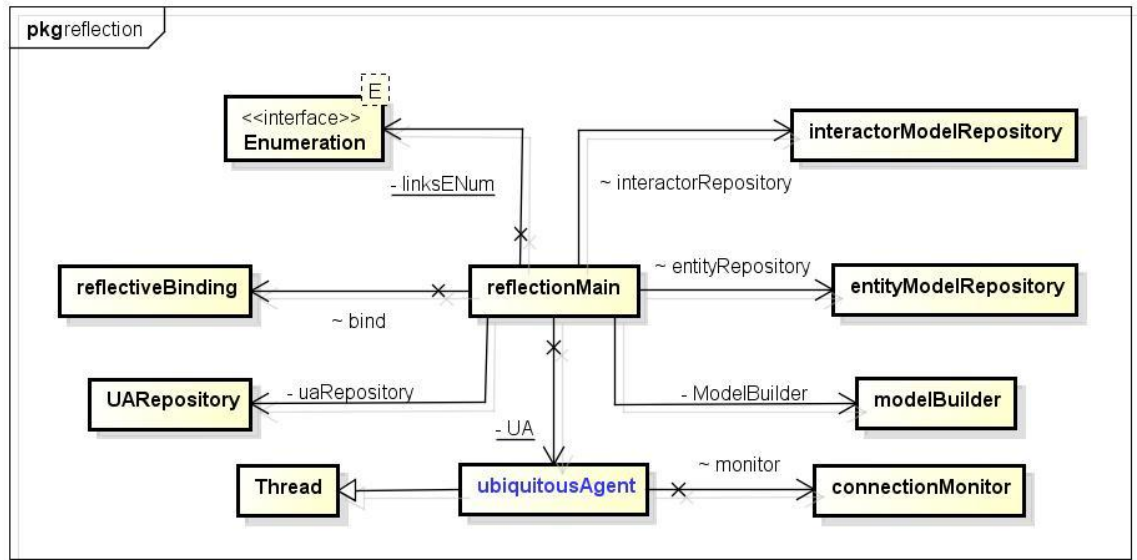


Diagram 5-6. UML diagram of the “reflection” package.

The “reflectionMain” class implements the entry point to the reflection layer. It deals with the dynamic creation of new models and UA sessions. It implements continuous loop functionality for receiving and managing notifications from the base layer of eVATAR using the “eventProcess” function (see Diagram B-16 in appendix B-3). In particular, it handles notifications about new BIL descriptions of agent/avatar sensors and actuators being added to the registration repository (see “registrationManager” in 5.1.2) by creating new interactors and entity models if they have not already been created (function “createModel” in class diagram of appendix B-3).

The “createModel” function of “reflectionMain” uses the “modelBuilder” class. The latter implements the “CreateInteractorModel” function of Listing 4-12 that takes a JAVA string containing the interactor metadata as an input and returns an interactor model object. The interactor model needs to be associated with an entity model. To achieve this, the “createModel” function calls the

“create_updateEntityModel” function³. In particular this function uses the interactor metadata description that is contained in a JAVA string as input and returns the entity model for it. If one does not exist it will create it and store it in the entity model repository data structure.

The “entityModelRepository” class⁴ implements a hash table structure and it is used by eVATAR as a data storage component containing entity model objects. It also provides with functions for updating and accessing the table. The “interactorModelRepository” class implements a hash table structure that is used as a data storage component containing interactor model objects⁵. This class also offers functions for updating and accessing the interactor model object storage. The “reflectionMain” creates both an entity and an interactor model repository that will be updated by the model creation function (“createModel”).

After creating the models, the “eventProcess” function checks whether the entities have been fully registered (see “requirements” in 3.3.2) with eVATAR. It achieves this by comparing the list of identifiers for registered interactors (that belong to the particular agent/avatar body) with the list of the identifiers for all the interactors (again that belong to the particular agent/avatar body) that are expected to register with eVATAR. As we saw in 5.2.1, both lists are stored in the entity model objects (following the specification of Listing 4-6 in chapter 4). If an entity is fully registered the “eventProcess” will start the binding process by searching for compatible entity model bodies within the entity model repository.

The “eventProcess” uses the “reflectiveBinding” class (Diagram 5-6) to implement the binding functionality that was illustrated in Flowchart 2 and can be found in the requirements section of chapter 3. Binding is a dynamic process taking place during runtime. As we saw in chapter 4, we call it reflective

³ It implements the function “ent_model_get_model” that was specified in Listing 4-9.

⁴ Listing 4-6 specifies the “ent_model_model_list” type that we implement in EVATAR with the “entityModelRepository” class.

⁵ It also implements the “intr_model_by_id” function that is specified in *Listing 4-15* and is used for accessing the hash storage to get an interactor model object based on its identifier.

because the binding in eVATAR differs from common SOA implementations in which the binding is taking place between service providers and consumers. The binding in eVATAR is between agent and avatar bodies which are identified sets of service consumers and sets of service providers.⁶

In the Z-Notation specification of reflective binding (section 4.1.7) we specified the third criterion of compatibility which is that compatible interactor service metadata descriptions should describe the same functionality and the way this comparison is achieved is a matter of implementation. For the specification we symbolized metadata description compatibility with the equality symbol “=”. In eVATAR we implement this by comparing the “Attribute” tags in the “Service” elements of the BIL descriptions that are stored as the “service description” in the metadata layer of interactor models. The tags should have the exact same values in compatible descriptions. As soon as a binding is created between an agent and an avatar model, they both cease to be available. Thus it won’t be possible for two or more agent models to link to the same avatar model.

The result of the binding is the creation of a UA object⁷ using the “ubiquitousAgent” class of Diagram 5-6. It stores the linking information between an agent model, an avatar model and their linked interactor models (“linkTable”). It implements a JAVA hash table that maps agent to avatar interactors. All UA objects are stored and managed by the UA repository object (“UARepository” class) that is also implemented as a hash table.

The “messageMediatorLoop” function of a UA object (see Diagram B-21) is run on a separate thread of program execution that loops reading messages from every outgoing message queue of the participating interactor models and it applies them to the incoming queues of the interactor models that are bound to them. We remind that these are the base layer queues for the connections with

⁶ The “reflectiveBinding” class implements the binding algorithm that was described using the Z-notation in Listing 4-21 (function “discoverAndBindBodies”). As we can see in the class diagram of the “reflectiveBinding” class in Appendix B-3, it also implements the support functions “initDiscoveryChecks”, “createBinding”, “createUA” that were specified in Listings Listing 4-16, Listing 4-18 and Listing 4-20.

⁷ UA objects implement the “MediateMessages” functionality that was specified in Listing 4-23.

the distributed interactor software that are accessed via the interactor models. This is how eVATAR implements the absorption and reification concepts of the reflective middleware (see 2.1). The following sample program code illustrates the main logic behind the implementation of the “messageMediatorLoop” thread.

```

while (RUNNING) {
    // the set of the two entity models participating in the UA
    entityModel[] models = {entity_1_model, entity_2_model};

    // For every participating entity model
    for (entityModel entityModel: models) {

        // The interactor identifiers for the entity model in the
        // list
        String [] interactors = entityModel.myInteractors();

        // For every interactor model identifier
        for (int i = 0; i < interactors.length; i++) {
            // get the corresponding interactor model
            interactorModel =
            entityModel.getInteractor(interactors[i]);

            // read the next message from the incoming message queue
            message = interactorModel.incomingQueue.nextMessage();

            // if the message is not null and it is a BIL message
            if (isMessage(message) == true) {

                //get the interactor model that is bound to it
                boundInteractorModel = linkTable.get(interactors[i]);

                // place the message to the outgoing message
                // queue of the bound interactor model.
                boundInteractorModel.outgoingQueue.add(message);
            }
        }
    }
}

```

Figure 5-6. Sample code describing the mediation within a Ubiquitous Agent.

The “messageMediatorLoop” implements a “model-mediator” in the sense that it mediates messages between interactor models. The latter receive the messages from the agent/avatar interactors that they model. If one mediator has a problem for example if a communication protocol fails, or if an agent or

an avatar interactor hangs on a blocking TCP connection it will not affect other mediators of other UAs.

For the duration of a UA session between an agent and an avatar, all messages are switched (forwarded towards the correct recipient) based on the messages' header that contains the identifier of the sender interactor (see interactor model metadata). The UA object will use this identifier with the linking table that will return the recipients identifier. This means that no processing of the body of the BIL XML message is required to perform the switching.

Finally, the "connectionMonitor" class is responsible for monitoring existing UA sessions. Every UA is monitored by a connection monitor object. If there is inactivity in a connection within a UA for a pre-defined period of time, the "connectionMonitor" object will test (via the connection server functionality of the base layer ⁸) whether the meta-interactor is reachable within the network. If the test fails, eVATAR will assume that the corresponding connection is lost and it will release all the memory and processing resources that are consumed for supporting the particular broken connection.

5.3. Summary

This chapter described the implementation of the eVATAR middleware. The implementation followed the Z-Notation framework of chapter 4 and this way we satisfy objective II of the thesis (see section 1.3). A two layered approach was used with a base layer and a reflection-layer that uses a reflective model in order to reason about and monitor the base layer. As we have seen, while the reflection component plays a very important role in terms of dynamically creating and managing connections between agent and avatar bodies, it is not intrusive in terms of the message exchanges within the connection sessions, practically serving as a message routing component. The two layered

⁸ We call "public boolean isReachable(int timeout)". Typically it will use ICMP echo requests. See:

[https://docs.oracle.com/javase/1.5.0/docs/api/java/net/InetAddress.html#isReachable\(int\)](https://docs.oracle.com/javase/1.5.0/docs/api/java/net/InetAddress.html#isReachable(int))

implementation aimed to simplify and systematize the application of connecting software agents to avatar bodies that are sets of sensors and actuators in the physical world. The “Case Studies” and “Evaluation & Discussion” chapters will exemplify and discuss the application of eVATAR in practical settings.

6. CASE STUDIES

This chapter describes two case studies that are used to exemplify and evaluate the eVATAR framework presented so far in this thesis. The first case study is about an implementation of a simple security scenario with the GOLEM [82] MAS, eVATAR and real sensors and actuators embedded in a miniature model home environment. The second case study uses simulation software designed to evaluate a more complex smart home scenario using eVATAR focusing on its potential for becoming useful in everyday life situations. This chapter is divided into two sections, one for each case study and each section concludes with an evaluation and the lessons learned about using eVATAR.

6.1. Miniature RoboHome

This case study revolves around the development of a miniature smart home, controlled by an intelligent agent that monitors interactions in the home to promote safety (e.g. from unauthorised intrusion). In this case study we sought a proof of concept using real sensors and actuators as opposed to simulated ones. The space and hardware limitations during the writing of the thesis led to the choice of a miniature smart home rather than a real instrumented home. An application using a real instrumented home with a human inhabitant is part of our future work plans as we will see in the future work section 8.2.

The objective of this case study is to provide with evidence for eVATAR in terms of satisfying the aims of the thesis by:

1. Integrating software agent functionality with physical sensors and actuators.
This should be demonstrated by enabling a software agent to interact with a set of physical sensors and actuators.
2. Implementing at least two scenarios each following the exact same method to enable a software agent to control sensors / actuators indicating a systematic way of achieving this.

3. Demonstrate transparency by using abstract interfaces at application level to integrate the agent with the sensor/actuator functionality without writing code for dealing with how the integrations are achieved (concealing dynamic discovery, binding and UA session functionality).
4. The two scenarios should be using different MAS platforms.
5. The two scenarios should use heterogeneous sensors, actuators and devices.

6.1.1. Scenario 1 – A Security Scenario using GOLEM

This scenario seeks to show how we can interface eVATAR and a MAS platform called GOLEM [82] that includes agents that are logic based and can provide interactions that are transparent to the user. A UA is going to implement a security scenario in a miniature house environment (Picture 6-1). The UA is going to make the house appear habited when the owner is away in order to deter burglars. The system runs autonomously and according to the scenario an observer should be able to interpret the light control management of the miniature smart home as an indication that it is habited when the house is empty. Furthermore, if the UA detects movement it will try to determine whether there is an unauthorized intrusion. If there is, it will collect photographic evidence and inform the owner of the house while attempting to discourage the intruder from remaining in the premises. Otherwise, it will welcome the entrant and perform light switch control based on movement within the miniature house.

The intrusion management is taking place every time a small toy-character is positioned inside the house. The observer of the experiment will be required to move the toy-person within the house, type verification passwords in a PC⁹ and interact with the miniature smart home.

⁹ The PC simulates a smartphone application that would interact with the home in a real world scenario.



Picture 6-1. Miniature Smart home before and after the UA has detected presence.

We implemented an agent in GOLEM and programmed it to bind to an avatar body within the miniature smart home environment. The following figure depicts the overall architecture of the system that consists of GOLEM, eVATAR and the miniature smart home environment. A video demonstrating the scenario can be found in YouTube [106].

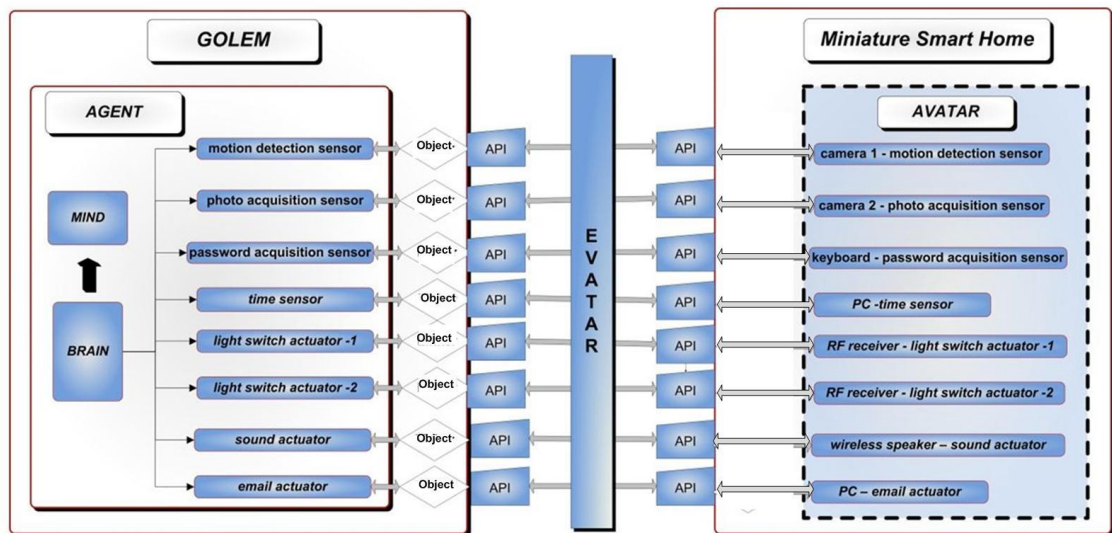


Figure 6-1. The architecture of the system that is described in this scenario.

In the following we will describe the avatar within the miniature smart home and the GOLEM agent while at the same time we will be showing how we used the proposed framework to enable the agent interactors to bind to the avatar interactors in order to form a ubiquitous agent.

A variety of wired and wireless sensors, actuators and devices was used to implement a low-cost network of sensors and actuators within the miniature house.

Avatar Interactors Function

wireless camera 1	sensor: motion detection
wireless camera 2	sensor: photo acquisition
Keyboard	sensor: password text acquisition
PC - computing device	sensor: physical world time acquisition
Waveman wireless receiver	actuator: light switch 1
Waveman wireless receiver	actuator : light switch 2
wireless speaker	actuator: voice\sound for user interaction, alarm and password request
PC - computing device	actuator: email notification

Table 6-1. "Miniature Smart Home" sensors and actuators.

The sensors and actuators in the above table feature interface software either written in JAVA for the purposes of the experiment or by using a custom SDK (Software Development Kit) such as in the case of the interface software for the Waveman receivers [127]. The software functionality is basic as all that is required is to pass control messages to the actuators and receive sensory data from the sensors. The interface software for every sensor and actuator is run on the networked computing device (PC) of Table 6-1.

Every sensor and actuator interface software uses the eVATAR API to interact with eVATAR. At the startup of the application, it uses the eVATAR API to start the TCP connection with eVATAR using the "apiInit" function (see Diagram B-11). This function also reads the configuration file and the BIL description of the particular interactor from the hard disk. These files are

created by the system developer. It then sends the BIL description of the particular interactor to eVATAR for registration purposes (see “apiRegister” function in the “evatarAPI” class that is described in Diagram B-11). Following the requirements of chapter 3 we had to create BIL descriptions for every agent and avatar interactor that will be participating in the system. For example in Figure 6-2 we see the BIL description of the password acquisition sensor (keyboard):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bilDescription InteractorID =" keyboardSensor1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bildescription.xsd">
<bilinfo>
<interactorType>SENSOR</interactorType>
<entityType>AVATAR</entityType>
<entityID> avatarPhysical </entityID>
<body>
<sensor> keyboardSensor1</sensor>
<sensor> timeSensor1</sensor>
<sensor> wirelessCamera1</sensor>
<sensor> wirelessCamera2</sensor>
<actuator> wirelessSpeaker1</actuator>
<actuator> wirelessReceiver1</actuator>
<actuator> wirelessReceiver2</actuator>
<actuator> emailActuator1</actuator>
</body>
<CommunicationType>REQUESTREPLY</CommunicationType>
<bindingType>agnostic</bindingType>
<target>na</target>
</bilinfo>
<Service>
<ServiceType>PROVIDER</ServiceType>
<Attribute>TEXT</Attribute>
<Attribute>ValueLong</Attribute>
</Service>
</bilDescription>
```

Figure 6-2. BIL Description of the keyboard sensor in the miniature smart home.

All avatar interactors have BIL descriptions like the one described above. The interface software calls the “apiGetBilMessageMetadata” function (see “evatarAPI” class in Diagram B-11) that creates and returns the BIL message

XML metadata that will be used for the messaging between the DIS and eVATAR. The BIL message metadata is created based on the xsd schema of Diagram 5-2. The interface software also calls the "apiGetBilObject" function from the "evatarAPI" class that returns a "bil" object that will enable it to edit the BIL message metadata during runtime transforming events e.g. sensory data from the interface software of the keyboard sensor into BIL messages. In the "keyboard sensor" example that implements the password acquisition functionality the BIL messages include the sensor and avatar identifiers, the type of the interactor and also a service element "Service" with the tags "TEXT" and "ValueLong" for the password and the time it was acquired (see 5.1.1). This BIL message that encompasses the sensory data comprising of the values in the "TEXT" and "ValueLong" tags will be sent to eVATAR.

Incoming messages from eVATAR are transformed into events by the interface software with the use of the eVATAR API. For example the interface software of the light switch will use the eVATAR API "bil" object to transform the incoming message into a command that controls the particular light switch in the physical environment.

We need to note at this point that the sensors and actuators do not necessarily need to be the traditional hardware interactors that are embedded in a physical environment. As we can see in Table 6-1, an avatar sensor can also be a piece of software that is external to the MAS. For example the physical world time acquisition sensor is run on a computing device and it is part of the avatar body. We consider sensors and actuators that are external to the MAS environment to be part of the miniature home sensor network even if they are also pure software.

In 3.2.1 we saw an overview of the GOLEM platform. For the purposes of the scenario implementation we created a GOLEM agent bearing sensors that sense events and pass the sensory data to the GOLEM agent mind and actuators enabling the latter to create events in the form of messages containing control commands. The interactors of GOLEM agents are implemented in Java.

The main difference in this implementation with the one of the avatars is that the GOLEM agent sensors and actuators (of the version of GOLEM that was used for this scenario) cannot interact with external to the agent environment resources such as the API directly. Instead they interact with the eVATAR API via GOLEM objects. In other words the agent interactors sent control requests to or sense events from GOLEM objects.

A GOLEM object uses its internal object (see 3.2.1) to interact with the eVATAR API (external resource) to send/receive data to/from eVATAR. It uses the API to transform received requests (e.g. in the form of control messages from agent actuators) into BIL messages towards the middleware. Furthermore, when a GOLEM object receives a message from eVATAR it will use the API to transform it and emit it as an event in the MAS environment. The latter will notify the appropriate agent sensors about the event. A sensing in GOLEM occurs when a sensor is notified by the agent environment about events that occur in it. For example the GOLEM object may emit events that encapsulate messages containing a "YES"/"NO" string that will be perceived by the motion detection agent sensor.

Every sensor subscribes to events from a unique GOLEM object. Similarly every actuator sends control requests (data structures containing values describing a command) exclusively to a unique GOLEM object. We created BIL description documents similar to the one in Figure 6-2 for every agent sensor and actuator. The eVATAR API enables the GOLEM objects to connect and interact with eVATAR in a similar way to the one of the physical interactors as described in the previous section through calling the same API functions and in the same order. With regards to participating in a system with eVATAR, an agent interactor is essentially a GOLEM sensor or actuator that features an exclusive one to one relationship with a GOLEM object that uses the eVATAR API to interact with eVATAR in order to exchange messages with a physical interactor that is bound to it.

The following table describes the interactors of the GOLEM agent.

GOLEM Agent Interactors

Function

GOLEM Agent Interactors	Function
motion detection sensor	senses events that are messages containing a "YES"/"NO" string of characters.
photo acquisition sensor	senses events that are messages containing a path to a photograph as a string of characters.
password acquisition sensor	senses events that contain a password as a string of characters.
clock sensor	senses events that contain a 64-bit integer number that represents the time in the physical world.
light switch actuator 1	creates an event containing a command in the form of a string of characters ("ON"\ "OFF").
light switch actuator 2	same as above.
sound actuator	creates an event containing a command as a string of characters with values describing a sound, speech, an alarm or a password request.
email notification actuator	creates an event containing a command as a string of characters that is typically an instruction to send an email.

Table 6-2. The GOLEM agent interactors.

We employ a basic reactive agent in GOLEM with the control cycle used in [83]. The cycle is specified in Prolog like syntax where " \leftarrow " is interpreted as "if", ",", as "logical and". Terms starting with small letters (e.g. "sense") denote

predicate names while terms starting with capital letters (e.g. “Percept”) denote variables. The GOLEM agent mind follows the cycle of execution that is described by the Prolog Program 1. We need to note here that the clock sensor of Table 6-1 is needed for the physical world time acquisition while the concept of time in GOLEM is based on the agent execution cycles.

```
cycle(Brain) ← sense(Brain, Percept),
                revise(Percept),
                choose(Action),
                execute(Brain, Action),
                cycle(Brain).
```

Prolog Program 1. The agent mind cycle.

As we can see above, there are four main stages in the cycle: “sense”, “revise”, “choose” and “execute”. The agent sensors sense their environment by creating what is referred to in the nomenclature of the GOLEM implementation in [82] as “percepts”. They are descriptions of the environment that can be understood and processed by GOLEM agents. The sensing from every sensor is mapped into a percept and received by the agent mind via the brain (see 3.2.1). This is performed in the “sense” stage of the agent cycle (see Prolog Program 1). The sensors update a queue with percepts. Following the agent implementation of [83], the sense function takes sensory data regarding events in the agent environment from a particular sensor in the list of sensors (see below “Sensors”) and transforms it into a percept within the agent mind as described by the following Prolog rule:

```
sense(Brain, Percept) ←
                        getSensors(Brain, Sensors),
                        getPercepts(Sensors, Percept).
```

Prolog Program 2. From sensing to a percept.

The mind this way knows at each cycle which sensor has perceived an event and the information associated with the particular event. The mind uses the

percepts to create an internal state of its view of the world and based on the percepts it will choose the next actions of the agent. The internal state is typically a set of variables describing what is perceived by the agents' sensors. The internal state is updated by the "revise" stage of the agent cycle. This is implemented using Prolog predicates, for example:

```
revise(do(golem_object, sensing_act, Event)) ←
    Event = avatar_event([Observation|Observations]),
    observation_effects([Observation|Observations]).
```

Prolog Program 3. Revise predicate example for managing a sensing act, i.e. an act that has been received by passive or active observation. GOLEM supports two additional acts: physical acts and speech acts. While sensing, an agent will need to also deal with speech acts, which for simplicity of presentation we omit, as it is defined similarly.

During the revise stage, a new internal state is created based on the previous ones and the newly sensed data. Above we see a "revise" predicate for updating the internal state of the agent using data from an event that was created by a GOLEM object. This data is essentially the sensory data from the avatar sensor that was received by the GOLEM object (that uses the API) via eVATAR and transformed into an event within the MAS environment in order to reach the corresponding agent sensor. The predicate `observation_effects/1` revises the state of the agent with the new observations. If the observation variable already exists, it overrides it. The new internal state of the agent is used by the "choose" stage (see below) of the agent cycle in order to enable it to decide the next actions. The choose function selects the action to be performed.

```
choose(Act) ←
    findall(select(Label, Act), select(Label, Act), Acts),
    higher_priority(Acts, Act).
```

Prolog Program 4. How to choose an action to execute (from [83]).

A logic-based approach was followed for activity recognition in order to enable the intelligent agents to select the next action based on environmental conditions. In this approach we keep track of all logically consistent explanations of the observed actions. This is done in GOLEM and specifically by defining this in the logic-based reasoning capabilities of the agents [82]. We consider all action selection rules. The basic activity recognition approach that was implemented for this scenario borrows elements from the works of Kautz in [36] and Artikis et al in [1], [2] with regards to what is considered an activity in the system and how to recognize it. However, we do not use time explicitly as they do, as we simply want to demonstrate feasibility of our approach according to the scenario specification rather than adhere to the principles of the generic activity recognition framework of [1] and [2].

We implement two types of perceived activities: simple and complex. A simple activity describes the perception of a singular event in an environment e.g. “motion detection” i.e. to detect movement in a room. Complex activities are combinations of simple ones. E.g. the simple activity “motion detection” combined with the simple activity “false password input” would result to the implicit recognition of the more complex activity “intrusion detection”. The logic-based rule in Prolog Program 5 describes the above example. It shows the way an agent in GOLEM selects actions in terms of what the agent has just observed. We can see that the agent has perceived two simple activities, which when combined can be used to describe how the agent should respond using a condition action rule of the form:

```
select(home_monitoring, sound(alarm, setVolume(high)) ←
    detected(motion(Activity1)),
    detected(authorisation(Activity2)),
    status_of(Activity2, failed).
```

Prolog Program 5. Selection rule example for the “choose” stage.

The above “select” form is interpreted as in trying to respond to a situation of detecting an “intrusion” which is used as a label for an action selection rule to

“sound alarm” if a motion has been detected and an authorisation request has taken place but failed. For more details on how to program GOLEM agents in the specific control framework the interested reader is referred to [82] and [83].

We implemented the scenario using the GOLEM agent and avatar that were described above. In the setting of this scenario, eVATAR is also running on the networked PC of Table 6-1. All agent and avatar interactors of Table 6-2 and Table 6-1 register to eVATAR and the latter links them to create a ubiquitous agent. Thus, the ubiquitous agent implements the security scenario using the logic based approach for activity recognition and action that was described in the particular section. In appendix C-4 we can see a description of the activities that are recognized and the UAs’ reactions to them in conjunction with the hardware used. The video in [106] provides with a better overview of the implementation of the security scenario.

6.1.2. Scenario 2 – A Simple Scenario Using JADE

We also implemented a JADE [25] agent and enabled it to interact with the Miniature RoboHome setting of Picture 6-1. We implemented a simple scenario in which a UA performs light switch control based on sensing presence within the miniature house. In particular, when we position a small toy-character inside the house it will switch the light on for 10 seconds. The observer of the experiment will be required to move the toy-person in the house.

The following table presents the hardware used to construct the avatar of the JADE agent within the miniature smart home.

Avatar Interactors	Function
wireless camera	sensor: motion detection
Waveman wireless receiver	actuator: light switch

Table 6-3. Avatar interactors for the second scenario.

Similarly to the previous scenario, BIL describes the sensor and the actuator while their interface software uses the eVATAR API to interact with eVATAR. For example in Figure 6-3 we see the BIL description of the motion detection sensor (camera):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bilDescription InteractorID =" wirelessCamera1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bildescription.xsd">
<bilinfo>
<interactorType>SENSOR</interactorType>
<entityType>AVATAR</entityType>
<entityID> avatarPhysical2 </entityID>
<body>
<sensor>wirelessCamera1</sensor>
<actuator> wirelessReceiver1</actuator>
</body>
<CommunicationType>REQUESTREPLY</CommunicationType>
<bindingType>agnostic</bindingType>
<target>na</target>
</bilinfo>
<Service>
<ServiceType>PROVIDER</ServiceType>
<Attribute>TEXT</Attribute>
</Service>
</bilDescription>
```

Figure 6-3. BIL Description of the motion detection sensor in the miniature smart home.

The light switch actuator also has a BIL description like the one described above. At the startup of the application, the avatar interactor interface software uses the eVATAR API to start the TCP connection with eVATAR by calling the “apiInit” function (see Diagram B-11) that also reads the configuration file and the BIL description of each interactor from the hard disk. Assuming the role of the system developers we created these files. Then, as in the previous scenario, the interface software sends the BIL description of the particular interactor to eVATAR for registration purposes (see “apiRegister” function in the “evatarAPI” class that is described in Diagram B-11).

The interface software calls the “apiGetBilMessageMetadata” function (see “evatarAPI” class in Diagram B-11) that creates and returns the BIL message XML metadata that will be used for the messaging between the DIS and eVATAR. It also calls the “apiGetBilObject” function from the “evatarAPI” class that returns a “bil” object that will enable it to edit the BIL message metadata during runtime transforming events e.g. sensory data from the interface software of the camera sensor into BIL messages. In the “wireless camera sensor” example that implements the motion detection functionality the BIL messages include the sensor and avatar identifiers, the type of the interactor and also a service element “Service” with the tag “TEXT” for the values “DETECTED” and “NOT_DETECTED” (see 5.1.1). This BIL message will be sent to eVATAR. The interface software of the light switch will use the eVATAR API “bil” object to transform the incoming message into a command that controls the light switch.

We also implemented a JADE agent with the following interactors:

JADE Agent Interactors	Function
motion detection sensor	senses events that are messages containing a “DETECTED” / “NOT_DETECTED” string of characters.
light switch actuator 1	creates an event containing a command in the form of a string of characters (“ON”\“OFF”).

Table 6-4. The JADE agent interactors.

JADE applications are implemented in JAVA. In our approach the piece of code in a JADE agent implementation that uses the eVATAR API to receive data from eVATAR is called a sensor and similarly an actuator is a piece of code that uses the API to send a control message to eVATAR. We described the

interactors of the JADE agent using BIL. For example, the BIL description of the JADE agent sensor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bilDescription InteractorID =" agentMotionSensor2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bildescription.xsd">
<bilinfo>
<interactorType>SENSOR</interactorType>
<entityType>AGENT</entityType>
<entityID> JadeAgent1 </entityID>
<body>
<sensor> agentMotionSensor2</sensor>
<actuator> agentLightActuator2</actuator>
</body>
<CommunicationType>REQUESTREPLY</CommunicationType>
<bindingType>agnostic</bindingType>
<target>na</target>
</bilinfo>
<Service>
<ServiceType>CONSUMER</ServiceType>
<Attribute>TEXT</Attribute>
</Service>
</bilDescription>
```

Figure 6-4. BIL Description of the motion detection sensor of the JADE agent.

Our agent features two cyclic behaviours (atomic behaviours that must be executed forever [31]), one for sending light switch control messages and one for reading the input from the motion detection sensor. It is a simple reactive agent implementation that essentially switches the light on for ten seconds if it detects motion in the entrance hall of the miniature smart home by setting a timer on. In the end of the timer it will switch the light off. Each behaviour uses an eVATAR API object to interact with eVATAR. The following JAVA code sample illustrates the implementation of the sensing behaviour in the JADE agent:

```

static evatarAPI api_ms = new evatarAPI();

...{

    api_ms.config.setConfigPath (configPath);
    api_ms.config.setBilDescriptionPath (motionSensorBILPath);

    /* Initialize the API, get the BIL description and connect to eVATAR.
       eVATAR then will start the binding process */
    api_ms.apiInit();

    /* Add the CyclicBehaviour for sensing. */
    addBehaviour(new CyclicBehaviour(this)
    {

        /* The motion sensor of the JADE agent */
        public void action()
        {

            if(motion_sensed == false)
            {
                System.out.println("Sensing motion");

                /* The motion sensor software using the API to receive a message.
                   If the binding process is complete and the agent is bound to an
                   avatar body it should start receiving valid messages */
                String receivedMessage = api_ms.apiReceiveMessage();

                /* If the received message indicates motion detection... */
                String txtAttr =
                    api_ms.bil.getTextAttributeValueFromBilMessage(receivedMessage);

                if (txtAttr.equals ("DETECTED"))
                {
                    /* update internal variable that a motion was sensed */
                    motion_sensed = true;
                    timer = 0;
                }
            }
        }
    } );
}

```

Figure 6-5. The sensing behaviour of the JADE agent.

And the acting behaviour that performs the light control management:

```
static evatarAPI api_ls = new evatarAPI();

...{

api_ls.config.setConfigPath (configPath);
api_ls.config.setBilDescriptionPath (lightSwitchBILPath);

/* Initialize the API, get the BIL description and connect to eVATAR.
   eVATAR then will start the binding process */
api_ls.apiInit ();

/* Add the CyclicBehaviour for switching the light on or OFF
   If motion has been sensed switch the light ON for 10 seconds. */
addBehaviour (new CyclicBehaviour(this) {

/* The light switch actuator */
public void action()
{
    String bilMsgMetadata = api_ls.apiGetBilMessageMetadata ();

    /* The light switch actuator of the JADE agent using the eVATAR API
       to send a control command. The internal variable motion_sensed can only
       be true if the binding process is complete and the agent is bound to an
       avatar body that has sent a valid BIL message that a motion has been
       sensed */
    if (motion_sensed == true && timer == 0)
    {
        System.out.println("Switching the light ON"+ motion_sensed);
        String to_Send =
        api_ls.bil.setTextAttributeInBilMessage(bilMsgMetadata, "ON");
        api_ls.apiSendMessage(to_Send);
    }
    else if(timer > 10)
    {
        System.out.println("Switching the light OFF"+ motion_sensed);
        String to_Send =
        api_ls.bil.setTextAttributeInBilMessage(bilMsgMetadata, "OFF");
        api_ls.apiSendMessage(to_Send);
        motion_sensed = false;
    }
    timer++;
}
});
}
```

Figure 6-6. The acting behaviour of the JADE agent.

For the purposes of the scenario implementation the sensor behaviour senses motion events and uses the sensory data to update the internal variables of the agent that will be used by the light switch actuator behaviour enabling the latter

to create events in the form of messages containing control commands. The sensor of the sensing behaviour is the “action()” method that performs active sensing by requesting sensory data via the eVATAR API. The actuator is the “action()” method that uses the API to send a control message with the indication “ON” or “OFF”.

In the setting of this scenario, eVATAR is running on the networked PC of Table 6-1. All agent and avatar interactors of Table 6-3 and Table 6-4 register to eVATAR and the latter links them to create a ubiquitous agent. Thus, the ubiquitous agent implements the light control management scenario.

6.1.3. Discussion

This case study provides with evidence for the eVATAR framework in terms of satisfying the aims of the thesis. Specifically:

1. In scenarios 1 and 2 we provided with evidence with regards to integrating software agent functionality with physical sensors and actuators by enabling a GOLEM agent and a JADE agent respectively to interact with a set of physical sensors and actuators within a miniature smart home.
2. In both scenarios, we followed the same systematic method to integrate two different MAS platforms with a different set of sensors and actuators each time. This involved describing each interactor using BIL, call the eVATAR API function to initialize a connection with eVATAR and then use the API to interact with it. eVATAR internally in both scenarios performed the dynamic discovery of compatible agent and avatar bodies, the binding and the management of the message exchange between the bound to each-other interactors. This case study further demonstrated the generality of the approach.
3. We demonstrated transparency in the two scenarios by using abstract interfaces from the eVATAR API at application level to integrate the agent with the physical sensors/actuators. We abstract interactor functionality by describing it as a service provider or a consumer using BIL. The eVATAR

API uses the abstract XML descriptions of BIL to enable the communications with eVATAR and to achieve the integrations. In both scenarios we did not need to write code for dealing with how the integrations are achieved (concealing dynamic discovery, binding and UA session functionality) and this functionality was executed transparently.

4. The two scenarios used different MAS platforms (GOLEM and JADE).
5. Heterogeneity was demonstrated by using heterogeneous sensors, actuators and devices within the miniature smart home. We currently only support platforms that use JAVA as the API is implemented in JAVA. JAVA runs on any platform or O/S that has the JAVA virtual machine installed. The eVATAR API is designed to communicate with eVATAR using message exchanges over TCP/IP and therefore we can implement it using any platform, programming language and for any O/S that supports TCP/IP development (see future work section for our plans to implement different versions of the API). This way we can include heterogeneous devices that do not support JAVA. eVATAR on the other hand would not require any changes in order to be used by heterogeneous MAS platforms and SAN.

Interoperability and openness are hinted via the aggregation of a variety of heterogeneous sensors and actuators working together within the UA body. For example, a light switch actuator and a camera motion detector sensor can appear in the environment as working together in order to achieve something useful (for example to implement behaviours such as *“if motion is detected turn the light on”*).

On a different note, all pieces of hardware in Table 6-1 were low-cost, easy to acquire and did not feature on their own elements of intelligence. We demonstrate that integrating them with a GOLEM [82] agent via eVATAR enables them to collectively present smart home behaviour and implement an intelligent security scenario. There are several low-cost and popular wireless technologies that include simple sensors and actuators in order to make a house "smart" and "automated" such as Z-Wave [134] and ZigBee [133] among others.

However the low-cost hardware on its own allows applications presenting limited intelligence that would not be able to provide all the functionality needed for a scenario such as the first one of this case study.

Furthermore, by evaluating the proposed approach in terms of satisfying the aims of the thesis using an application that integrates the GOLEM MAS [82] and real sensors and actuators and an application using JADE [31], we satisfy objective III (see section 1.3). We infer from this observation that by using the Z-Notation framework of chapter 4 we specified the functionality of eVATAR, a middleware that fulfils the aims of the thesis. This way we also satisfy objective I of the thesis.

In this case study due to limited resources available during the writing of the thesis we used a small number of sensors, actuators and devices. The next case study was designed to evaluate eVATAR using more agents and multiple sensors/actuators.

6.2. Simulated RoboHome

We have provided with evidence for eVATAR in terms of satisfying the aims of the thesis. The next step was to provide with insight as to whether it can be used in a more complex scenario with multiple sensors and actuators. We developed the RoboHome simulation, a smart home simulation that can be used with eVATAR and a MAS that has the characteristics defined in section 3.2.1 to implement test applications. In many cases SAN, WSN and pervasive middleware are evaluated using custom simulation software. Examples of middleware that use custom simulations include MagnetOS [80], the cluster-based middleware approach of [98] and TinyDB [88]. This way developers test specific features of the middleware that are relevant to a specific type of application. These types of testing environments are very useful when researching and prototyping new approaches allowing for insight as to whether the middleware achieves its aims.

Nevertheless custom middleware are usually not useful in terms of comparing different middleware approaches due to the fact that they focus on different middleware characteristics and metrics. On the other hand generic testbeds such as TOSSIM [77] and Indriya [64] can be used by different middleware for evaluation. The problem with generic simulation platforms is that it is difficult to provide benchmarks for pervasive middleware frameworks due to the application specific nature of the latter. Such simulations are more appropriate for low-level middleware for WSN as they can adequately provide distribution performance information. Also, they commonly focus on sensors and sensor networks and not on actuators. Furthermore, the platform and programming language dependencies of generic middleware simulations and testbeds (in the case of TOSSIM [77] and Indriya [64] these are the TinyOS operating system and the C/C++ programming language) exclude middleware that do not support them. Currently eVATAR only uses a JAVA based API but as part of our future work plans we intend to implement the eVATAR API using C/C++. Therefore we opted for a custom simulation to evaluate eVATAR.

The objective of this case study is to implement a scenario using RoboHome and GOLEM that shows in simulation the potential of our approach in becoming useful in a person's every day activities.

6.2.1. The Scenario

The scenario revolves around the simulation of a specific period (a week) in the life of a working person living in a small flat. The scenario assumes that "the resident" (as we will be calling the simulated person) works five days a week and performs a number of activities when he is at home. These include laundry (in a semi-random manner as we will see in the following), watching movies and TV programs, sitting in the living room and listening to music, going out on a Saturday night and in general several typical activities of modern lifestyle.

According to the scenario, the simulation is run in two modes. In the first mode, an electricity consumption monitoring agent in the GOLEM MAS is linked via

eVATAR to sensors in the smart home in order to create a UA that will be monitoring the consumption during the simulated period. In the second mode, a second UA is created by linking an energy consumption conserving agent to an avatar in the smart home consisting of sensors and actuators. The two UAs will communicate at agent level in a way that they will monitor and save on the energy consumption using the actuators of the second UA. In appendices C-1 and C-2, we can see detailed descriptions of the avatars and agents that were implemented for the scenario. Before proceeding with the details of the implementation of the scenario, it would be useful to describe the RoboHome simulation.

6.2.2. The RoboHome Simulation

To implement the scenario we developed a smart home simulation called RoboHome which is a discrete event-based simulation (DES) [90] implemented in Java.

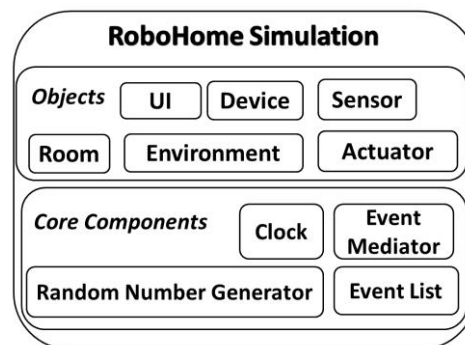
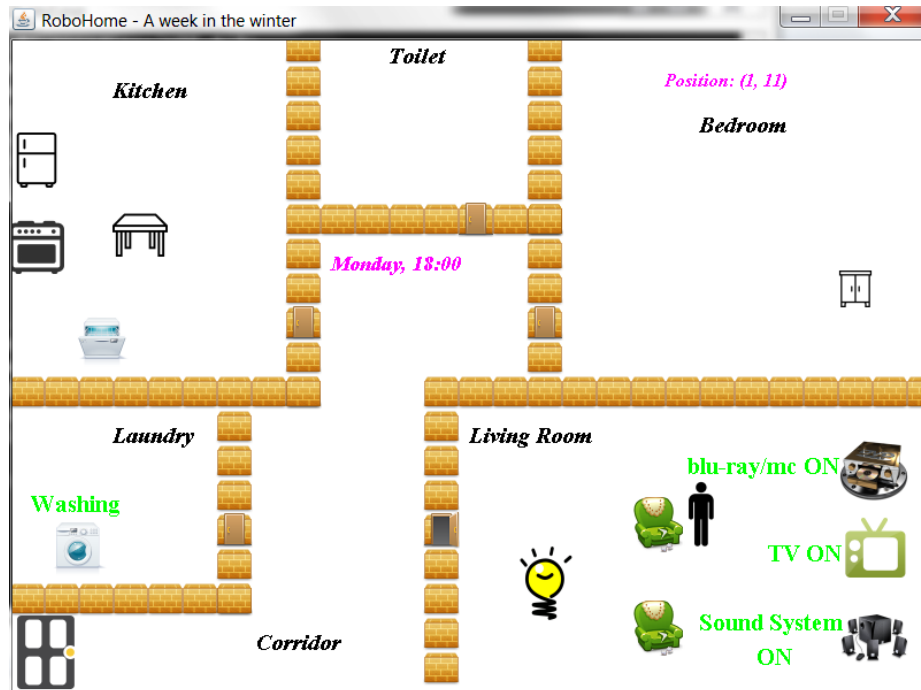


Figure 6-7. The RoboHome architecture reference model.

Figure 6-7 illustrates the main components of RoboHome. The current version of RoboHome that was used for evaluating eVATAR consists of a set of JAVA objects that model the smart home environment. The “environment”, “room”, “device”, “sensor” and “actuator” objects are created to model the smart home that is represented in a UI that was implemented using JAVA and depicted in the following picture:



Picture 6-2. The RoboHome GUI.

The smart home simulation model can be described as a collection of events. Each event models a change in the state of the above objects (see below for what we define as a “state” in RoboHome) and it schedules other events that are linked to this particular event. We will now describe the main concepts of the RoboHome DES.

The “Environment” software component of the smart home module (Figure 6-7) is a JAVA object implementing a virtual space where all other entities and objects in the simulation are created and reside. The development framework is implemented using the JAVA concept of inheritance [109].

In Figure 6-8 we can see how the various device subclasses are derived from the “Device” superclass. All subclasses are overriding [119] the turnOn(), turnOff() methods with their own underlying implementation. For example

turning on an oven and turning on a light have a different significance and implementation with regards to simulating electricity consumption as they start different electricity consumption counters e.g. an oven would consume more than a light bulb. Also, a subclass may have more methods as we can see in the TV device subclass that has a third method for setting the TV on a standby mode (and consuming electricity at a reduced rate).

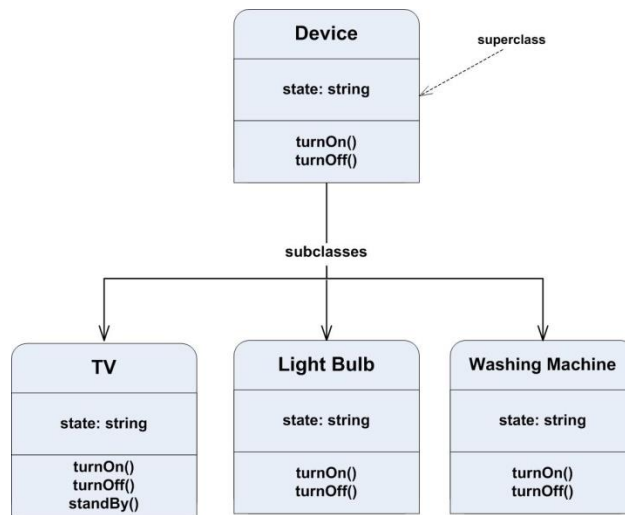


Figure 6-8. Subclasses of the "Device" superclass.

RoboHome and the objects it contains have a state which is a description of what is happening to them at a particular time. Time is implemented as a periodically increasing variable (see "clock" component in Figure 6-7) and the period of the time variable increment is intended to scale the time of the real world scenario that is simulated by RoboHome. The state is typically described by a single or a combination of descriptive variables. The state is altered by a set of interface methods that interact with these variables. For example the state of a device object such as the state of the TV features a variable with values "ON"/"OFF"/"STANDBY" and methods to interact with it (see "turnOn", "turnoff", "StandBy" in Figure 6-8). The main classes that are offered for creating subclasses and objects within the simulation environment are the:

- Room: creates sub--environments within the environment containing other objects.

- Device: objects simulating electrical devices such as a TV, a home cinema, a light bulb or a washing machine.

We call events any changes in the state of a variable belonging to a RoboHome JAVA object. Each event is associated with an event time and actions will be executed when the event occurs. Furthermore, the sensors are JAVA objects reading data about the state of the environment and device objects and the actuators JAVA objects altering the state of the environment and device objects by creating events. They read and alter the state variables of objects via the object methods (functions). They also use the API to make the connection to eVATAR possible. The system designer creates BIL descriptions for the smart home interactors. The sensor/actuator control software uses the eVATAR API to retrieve the BIL descriptions of their interactors and then use the metadata to register and interact with eVATAR (via the API).

Simulation events are created by the event list JAVA object (see reference model in Figure 6-7). Event lists are JAVA objects producing queues of different kinds of events and they can simulate humans producing events (e.g. the “resident” of the scenario that was described in 6.2.1). An event from the event list can also lead the creation of other events. For example, an event list object may create an event by directly having the “resident” enter a room (change the state of the room to “OCCUPIED”) which will lead to the creation of a second event from the UA that will use an actuator to change the state of the light to “ON”.

The event mediator JAVA object is a broker for matching the state changing requests from the event list and actuator objects to the state changing methods of the corresponding objects. Sensor objects also use the event mediator to query about changes in the state variables of the objects they observe. The mediator is implemented as a set of hash-tables containing pairs of objects in which: a) the first object is the creator of an event while the second object the recipient of it or b) the first object senses events (e.g. a sensor) created by the second one. This way we define which actuators act on which objects and which sensors sense which objects. Lastly, an object can create an event

targeting multiple other objects and similarly a sensor can observe multiple objects.¹⁰

The “Random Number Generator” component of Figure 6-7 affects the occurrence and the duration of events. The occurrence of an event is based on three factors: a) the existence of the event in the event queue (sequential list of events), b) the user pre-defined restrictions for the occurrence of this event (e.g. maximum washing up frequency per week) combined with the current state of the system (e.g. room occupancy status) and c) a consultation with the probability calculator functionality. The probability calculator is only consulted if both a) and b) have already been satisfied. It is part of the “Random Number Generator” component and provides with functionality that will reach to a decision with regards to the creation of an event while taking into account of a weight that has been pre-programmed for the particular event and by using a discrete uniform distribution. The weights are applied when the event list object is created using the appropriate template (more details in Appendix C-5).

All events have a duration (in clock cycles) which is decided based on: a) user pre-defined restrictions for the duration of this event and b) the random duration function (for more details see Appendix C-6). The duration restrictions (upper and lower limits) are pre-set for each event. For example the “night sleeping event” may last between 6 and 10 hours. The random duration functionality which is also provided by the “Random Number Generator” component will select a random number within the limits indicating the exact duration.

The RoboHome environment, room, device, sensor and actuator objects are created and organized as per the UI snapshot of Picture 6-2. To create a new simulation based on a particular scenario, the system designer creates a simulation script that is translated as an event list. Furthermore, RoboHome provides the capability to use the sensor and actuator superclasses to create sensor and actuator objects that do not participate in the simulation of Picture

¹⁰ Both cases are implemented with multiple entries in the hash-table structures.

6-2. They can be described by the system designer using BIL in a way that they participate in an avatar body and use the eVATAR API to connect and interact with eVATAR. This way they can create UAs with agents and can be used for performance testing scenarios as we will see in chapter 7.

6.2.3. The Implementation of the Scenario

The scenario described in 6.2.1 was implemented using the RoboHome simulation. It is visualized in a GUI (Graphical User Interface) and in Picture 6-2 we saw an instance of it during a simulation run.

We used RoboHome to implement the avatars of appendix C-2. With regards to the GOLEM agents, we used the same implementation approach that was used for the agent that was described in 6.1.3. The agents follow the cycle that was described in Prolog Program 1, with the only difference being the set of goals that they pursue. They are also using a special type of acts in GOLEM, the “speech acts” that allow agents to exchange messages (and context) within the agent environment. The agents are described in C-2. The requirements of chapter 3 are also followed, similarly to the framework implementation in the first case study (section 6.1.1).

The setting of the scenario was described in 6.2.1. We run the simulation twice in order to monitor electricity consumption with and without intervention from energy saving agents (appendix C-2). The electricity consumption is essentially the collective consumption of all devices that are monitored by UAs. The wattage and its cost for every device has been approximated based on realistic expectations (see Figure C-0-1 in appendix C-3) and the same values and costs have been applied in both runs of the simulation.

The first time the electricity monitoring agent monitored only power consumption and it was not allowed to intervene in the everyday life of the user (event list object simulating the resident). In the second run, all agents were linked to avatar bodies within the Smart Home and they were allowed to intervene. Particularly, in the second run the consumption improving agent

described in appendix C-2 applied its policies to the smart home by switching on and off devices appropriately. For example it would switch the lights or devices off when the user would forget to do so (most commonly when leaving the room for a considerable amount of time. Other policies/goals of the agent are described in appendix C-2). In order to achieve this, it communicated with the first agent that provided with contextual information regarding which devices were consuming electricity at a given time and location.

The electricity consumption monitoring agent provides this information to the consumption improving agent using agent to agent communication protocols within GOLEM (“speech acts” in [82]). It sends messages containing information regarding: a) identification of devices consuming electricity, b) identification of the location of these devices, c) information about the time that the consumption started and the time it ended and d) the power consumption specification of the relevant devices and the cost (some devices consume more than others and at a different cost). This is also related to the time of consumption (day and night cost rates). In Figure 6-9 and Figure 6-10 we can see the differences in electricity consumption and cost over a period of approximately 7 days with and without intervention.

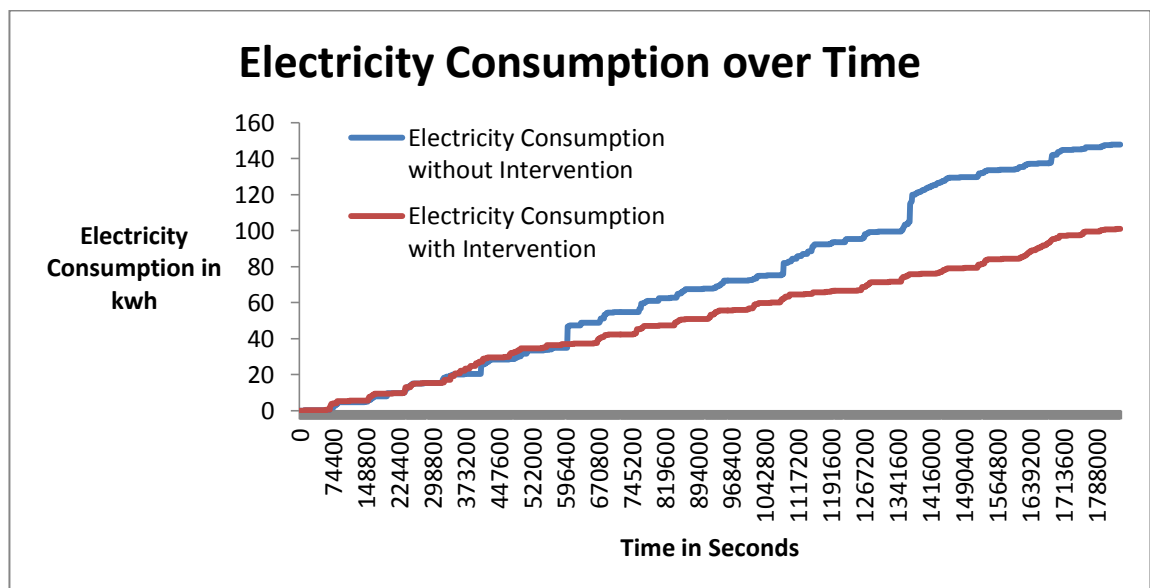


Figure 6-9. Electricity Consumption with and without intervention from UAs. The time in seconds refers to the scaled real time.

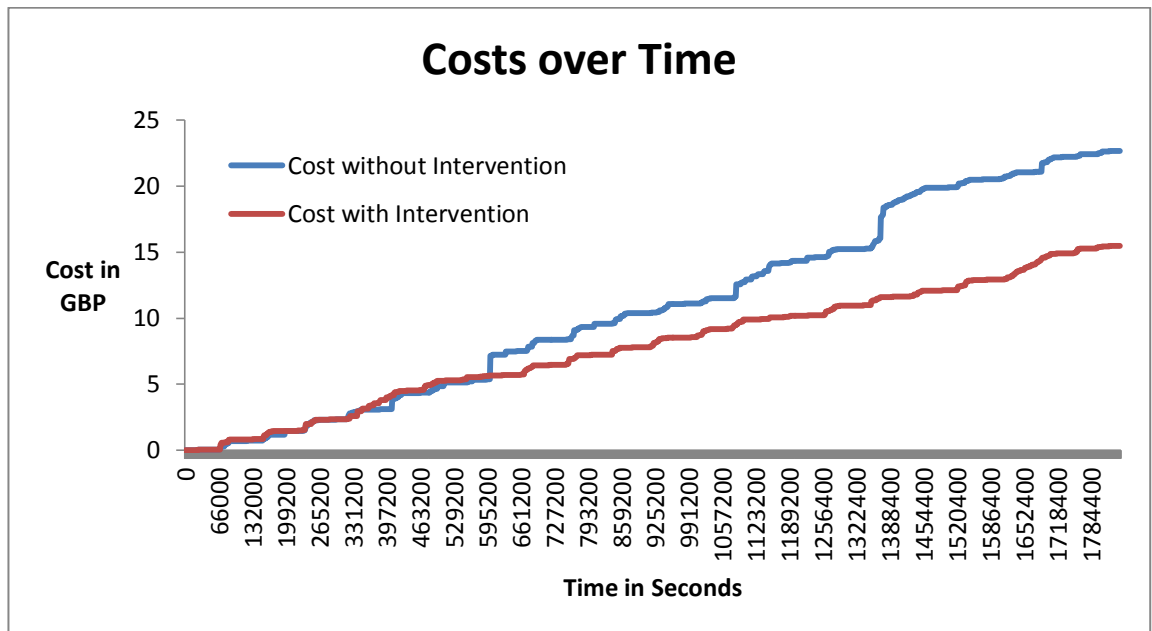


Figure 6-10. Costs with and without intervention from UAs.

The improvement in terms of cost and energy consumption when using eVATAR and Ubiquitous Agents is illustrated by the above diagrams.

6.2.4. Discussion

The simulation demonstrated that the behaviour of simple devices in the UA body is affected by the collective behaviours and context of other devices within the same UA body as well as from context stemming from other UAs. The software agents and their associated UAs, can exchange information within the MAS. The energy consumption monitoring agent was communicating and cooperating with the energy consumption improving agent by means off the MAS environment. This way the energy consumption saving agent was enabled to make more informed decisions regarding its next actions. Thus, the devices belonging to the avatar bodies in the smart home appeared in simulation to present more intelligent and context aware behaviours when connected to agents via eVATAR. Taking this one step further, we can view all UAs as fully communicative nodes of an AmI system. In the RoboHome simulation the

simple devices are aggregated into UAs (nodes) transforming the simulated smart home into an AmI environment.

Finally, the scenario of the RoboHome simulation illustrated the potential of a system using MAS agents, eVATAR and a sensor/actuator network embedded in a home context for becoming useful in confronting everyday lives problems. In the specific scenario of the simulation, we saw that the proposed UAs have the potential of reducing electricity consumption in a smart home environment, given that the assumptions made in the simulation hold (see in Figure 6-9 and Figure 6-10).

In this section we satisfy objective IV of the thesis (see section 1.3) by creating a test suit consisting of a smart home simulation and GOLEM and by implementing a scenario that shows in simulation the potential of our approach in providing useful services in our everyday lives (in our case saving on electricity consumption).

6.3. Summary

This Chapter presented two case studies that illustrate the use of eVATAR. The first case study provided with proof of concept that eVATAR can be used for the creation of UAs in an application with commercial sensors and actuators and MAS such as the GOLEM and the JADE platforms. It also provided with an evaluation of eVATAR in terms of satisfying the aims of the thesis. The second case study revolved around the RoboHome simulation and a MAS being used to implement an electricity consumption improvement in a smart home scenario. The simulation was used to investigate the potential of architectures that use eVATAR for becoming useful in a person's home environment (e.g. saving on electricity consumption).

7. EVALUATION & DISCUSSION

This chapter starts with an evaluation of eVATAR in terms of performance and then continues with a discussion regarding failure handling in systems that use it. The last section of this chapter compares eVATAR against the middleware platforms that were discussed in the “Background” chapter 2 with regards to achieving the aims of the thesis.

7.1. Performance & Scalability Testing

Performance in systems that use eVATAR is critical as such systems are typically real-time constrained by operational deadlines. For instance consider the simple scenario of a UA programmed to react to human presence in a room by switching a light on. All the message exchanges between the agent and physical sensors and actuators that are involved in this scenario need to take place within specific deadlines. Missing any of them would lead to a failure in meeting the constraints and to potentially undesired behaviour e.g. the human remaining in the dark for a prolonged period of time. In the following we evaluate eVATARS’ performance while the number of the connections (and UA sessions) it serves increases in a variety of hardware settings. The goal is to identify situations in which the real time constraints are not met and also discuss ways to avoid them.

7.1.1. Latency in eVATAR

The main objective of this evaluation is to monitor communication latency and identify the implications of scaling a system that uses eVATAR. Such systems scale by adding more agent and avatar interactors and thus creating more connections to eVATAR.

Network latency can be measured either “one way” as the time it takes for a packet to arrive from the source to the destination receiving it, or as “round trip” delay time. The main problem with following the “one way” approach is that we would need a global time. As we saw the two environments (MAS and physical world system) that are integrated by using eVATAR do not necessarily use a global time approach and they may feature different time systems. Their interactions are managed and synchronized by eVATAR. Therefore, we decided to collect “round trip” latency data as the time between sending a message and receiving a response. In our tests with eVATAR, the latency time is calculated as:

$$\text{Equation 1: } L(m1, m2) = T_{ag \rightarrow av}(m1) + T_{av \rightarrow ag}(m2)$$

Where:

L is the Latency.

m1 is the message that the agent interactor sends to the avatar interactor.

m2 is the message that the avatar interactor returns to the agent interactor.

T_{ag→av}(m1) is the time for message m1 to reach the avatar interactor (from the agent interactor).

T_{av→ag}(m2) is the time for message m2 to reach the agent interactor (from the avatar interactor).

The individual times for sending the messages from the agent interactor to the avatar one and vice versa are calculated as follows:

$$\text{Equation 2: } T_{ag \rightarrow av}(m1) = T_{ag \rightarrow eVATAR}(m1) + T_{eVATAR}(m1) + T_{eVATAR \rightarrow av}(m1)$$

and

$$\text{Equation 3: } T_{av \rightarrow ag}(m2) = T_{av \rightarrow eVATAR}(m2) + T_{eVATAR}(m2) + T_{eVATAR \rightarrow ag}(m2)$$

Where:

T_{ag→eVATAR}(m) is the time for a message m from the agent interactor to reach eVATAR.

T_{eVATAR}(m) is the time that eVATAR takes to process a message m.

$T_{\text{eVATAR} \rightarrow \text{av}}(\mathbf{m})$ is the time for a message from eVATAR to reach the avatar interactor.

$T_{\text{av} \rightarrow \text{eVATAR}}(\mathbf{m})$ is the time for a message from the avatar interactor to reach eVATAR.

$T_{\text{eVATAR} \rightarrow \text{ag}}(\mathbf{m})$ is the time for a message from eVATAR to reach the agent interactor.

7.1.2. Test

The base layer of eVATAR implements a TCP server allowing connections (see chapter 5) with the agent and avatar interactor software. Each connection requires a separate thread of execution that consumes memory and processing resources. GOLEM and the RoboHome simulation (see chapter 6) were used as an agent and avatar generator for our tests. The scaling of the system is essentially achieved by increasing the number of connected interactors. In our tests, we achieve this by gradually adding new agents and avatars bearing more interactors resulting in the creation of more eVATAR threads serving the connections with the interactor software. Latency (see below) is monitored and recorded during the scaling of the system.

The setting of the tests involves two networked computers, one that runs GOLEM and the RoboHome smart home simulation and the other running eVATAR. A number of tests have been performed, with eVATAR running on a variety of machines featuring different levels of memory and processing capabilities. The test scenario is the same in all tests; the only thing that changes is the hardware that runs eVATAR.

The test starts with one agent and one avatar carrying one actuator each and connecting to eVATAR. After they are registered and bound, the agent actuator starts sending BIL messages to the avatar actuator which in its turn returns them back as feedback messages with no processing. This way there is no delay on the recipient side. The size of each message is 503 bytes. According to the latency equation for eVATAR:

$$L(m) = T_{ag \rightarrow av}(m_sent) + T_{av \rightarrow ag}(m_rec)$$

In the case of our tests $m_sent \approx m_rec$ meaning that the same size messages are sent and received for measuring the latency. The frequency of sending the messages is a message for every 5 milliseconds plus the latency of the previous message. Gradually and every 1 minute a new agent and a new avatar are added to the system each of them bearing 30 actuators. After registering with eVATAR and becoming bound, they would start sending messages similarly to the first actuator and again with a frequency of 1 message per 5 milliseconds plus the latency of the previous message. This process will go on with new agents and avatars bearing 30 actuators each being added every minute until the system becomes unstable indicating the limitations of the system for the particular hardware. In all tests the network speed was the same at 100mb\sec. The different machines running eVATAR on each test:

- 1) "common" student laptop with CPU: Intel Core i5-2520M @ 2.50GHz, RAM: 4gb, OS: Windows 7
- 2) "low - end" computer with CPU: Intel Atom @ 1GHz, RAM: 2gb, OS: Windows 7
- 3) "high - end" computer with CPU: Intel Core i7-2670QM @ 2.20GHz, RAM: 8gb, OS: Windows 7

The tests measure latency which is recorded in relation to time and stored in a file.

7.1.3. Results

The first test was run 50 times producing as many datasets for latency. Figure 7-1 shows a representative chart from one of the runs. In all 50 test runs and in particular beyond a threshold of a total of 722 connections (between the 720th and the 730th second), the charts (just like the one of Figure 7-1) would start showing increased latencies and spikes. According to the test scenarios, the

system was scaled by connecting batches of interactors. The threshold in all 50 runs of the test was consistently reached when the same number of batches was connected (722 connections).

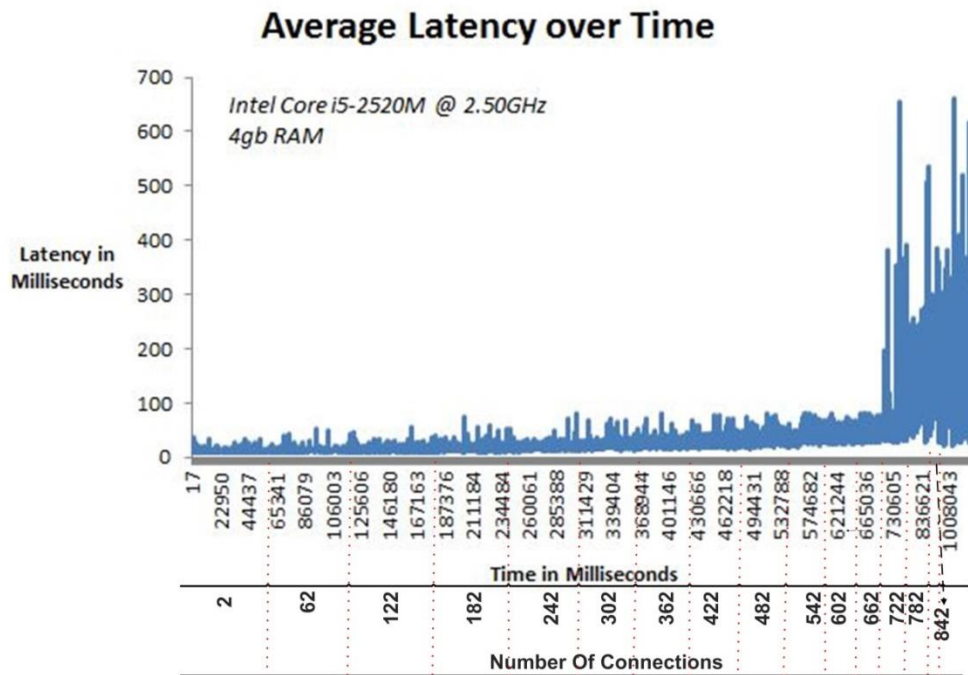


Figure 7-1. Using Equation 1 to create an average latency over time graph for the first piece of hardware. Appendix D-1 shows the numbers of agents, avatars and interactors that are being served by eVATAR as they increase during the execution of the test.

We calculate the mean (average) latency of every run of the test from the beginning until we reach the threshold number of connections beyond which the system becomes unstable with high latencies (722 connections). The goal is to quantify the amount of variation in the 50 mean latencies by calculating the standard deviation ([93] and [121]). In our controlled environment experiments we see that the Bell curve of the distribution of the data (Figure 7-2) is very steep meaning that the data has a small standard deviation and it is clustered around the mean (which in this case is the mean of the mean latencies). This implies that there is little dispersion in the set of mean latencies of the 50 tests until we reach the threshold. The main point though is that beyond this threshold in the number of connected interactors we see greater variances and instability.

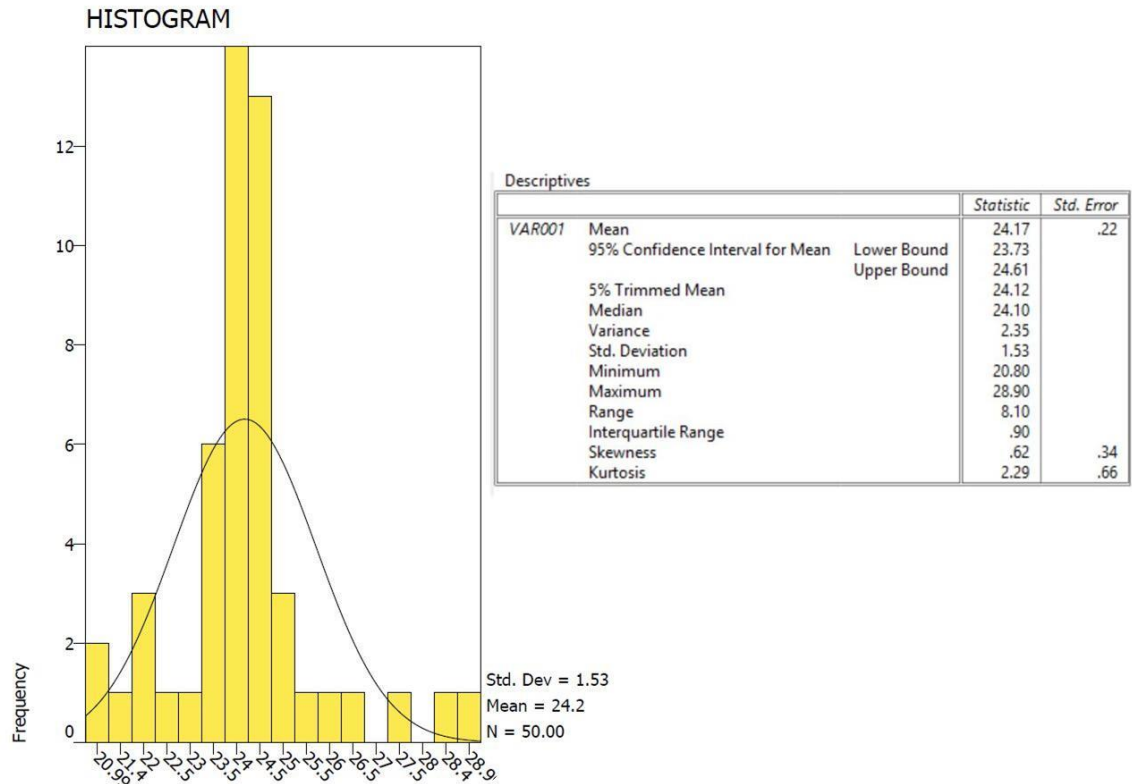


Figure 7-2. Bell curve of the distribution of the mean latencies. The mean here is the mean of the mean latencies. We see that most values are close to the mean.

For the second test, eVATAR was run on the “low-end” computer that was described in the previous section. The test was also run 50 times. We observe similar results to the ones from the first test with the main difference being that the middleware could accept less agent/avatar interactor connections. The system this time starts becoming unstable (in all 50 runs) beyond a threshold of a total of 482 connections (the 490th – 500th second). The third test was performed using the third computer that was described in the previous section. The test was again run 50 times. This time an average of a number around 1202 interactors had to connect before the latency started to significantly increase. In all tests the networking speed of the LAN (Local Area Network) remained the same and the same LAN router (with network speed at 100mb/sec) was used. The same tests were also carried by disabling the reflection layer of eVATAR and hardcoding the relationships between agent and “physical” interactors. Below we see a table describing all the experiments and results. Also, it would

be important to point out here that the average latency in Table 7-1 is the average (mean) of the mean latencies of all 50 test runs before the threshold of the number of connections is reached.

Hardware / Software Setting	Threshold (Number of connections)	Average (mean) of Latency Means before Threshold
CPU: Intel Core i5-2520M @ 2.50GHz, RAM: 4gb, OS: Windows 7 Reflection: YES	722	24.2 ms
CPU: Intel Core i5-2520M @ 2.50GHz, RAM: 4gb, OS: Windows 7 Reflection: NO	722	24.1 ms
CPU: Intel Atom @ 1GHz, RAM: 2gb, OS: Windows 7 Reflection: YES	482	31.85 ms
CPU: Intel Atom @ 1GHz, RAM: 2gb, OS: Windows 7 Reflection: NO	482	30.49 ms
CPU: Intel Core i7-2670QM @ 2.20GHz, RAM: 8gb, OS: Windows 7 Reflection: YES	1202	21.81 ms
CPU: Intel Core i7-2670QM @ 2.20GHz, RAM: 8gb, OS: Windows 7 Reflection: NO	1202	20.66 ms

Table 7-1. Test results with averages of latency means for the 50 runs of each test and the connection thresholds for the different hardware settings.

7.1.4. Discussion on Performance

The tests indicated a clear relationship between scalability, hardware and performance. When the number of connections to the middleware surpasses a certain threshold which varies in different settings and hardware, we observe greater latencies. Adding more connections to eVATAR would eventually

render it unstable. The question at this point would be whether the choice of having a multi-threaded, blocking server implementation providing a thread of execution for every connection was the right one.

Multi-threaded and blocking servers perform well with smaller numbers of connections (see metrics in Table 7-1) but are not as scalable [49] as single-threaded event driven servers e.g. servers implementing the JAVA Apache “Mina” framework [101] or “libuv” [111]. Event driven servers use a single thread to process messages from multiple connections. They do not bare the processing costs and memory overhead from having multiple threads and they provide better scalability even when handling thousands of concurrent connections. Scalability is critical in common web server applications as they are typically designed to manage thousands of connections simultaneously (see C10k problem [128]) that are usually short lived and deal with small requests. They need to serve as many concurrent connections as possible. This leads to an overhead as more connections typically compete for the same memory and processing resources. On the other hand transferring large files usually requires a dedicated connection, such as the ones offered by blocking and multi-threaded implementations.

eVATAR requires permanent dedicated connections for the sensor and actuator communications justifying the choice for a blocking, multi-threaded server. The principle of transmitting sensory data from a sensor to the middleware is not dissimilar to the one of transferring large files over a network. The applications that use eVATAR are more dependent on performance as it is offered by a dedicated thread rather than scalability. The number of connections per instance of eVATAR running, especially in smart home applications is considerably lower than the one of web servers (see scenarios in the case studies of chapter 6) but there are still limitations. Therefore, the challenge when designing systems that use eVATAR on a specific hardware platform, is how to avoid a situation where the real time constraints are not met adequately and also improve the performance of eVATAR.

Larger scale applications would require splitting the workload over multiple instances of the eVATAR middleware running on the same or on distributed machines in order to increase speed and scalability. On the other hand, this approach raises costs as it relies on using more hardware devices running eVATAR. The abundance of computing devices in modern households could alleviate the cost problem.

Finally, based on the test results of Table 7-1, disabling the reflection layer of eVATAR and hardcoding the connections would produce similar average latencies indicating that the overhead of the reflection functionality of eVATAR is insignificant. This behaviour was expected because while the reflection component is responsible for dynamically creating and monitoring connections between agent and avatar interactors, it is not intrusive in terms of the message exchanges within the UA sessions after they have been established. This way eVATAR reduces overhead and unnecessary processing that might have had a toll on connection performance. According to [33]: “a desirable middleware model provides transparency to the applications that want it and fine-grain control to the applications that need it”.

By using the RoboHome simulation to test the middleware’s performance we also satisfy objective V of the thesis (see section 1.3).

7.2. Failure Handling & Resilience

This section discusses resilience as the ability of systems that use eVATAR to cope with different levels of failure including failure at middleware and failure at agent/avatar level.

7.2.1. Failure at Agent & Avatar level

Applications that use eVATAR potentially include a variety of hardware and software components including agents, avatars, sensors, actuators and devices. Failures occur due to loss of connectivity, software crashes/bugs and broken hardware among other causes. eVATAR ensures that failure of a single

interactor does not mean failure of the whole UA as the rest of the linked interactors should remain unaffected within their individual connection threads. Despite the fact that we connect bodies, if a body member is broken it doesn't mean the whole body is broken. Similarly, if a set of interactors fails rendering an agent or an avatar and consequently the UA entity inoperative, the rest of the UAs should proceed with their tasks unaffected. On the other hand, if multiple UAs (or interactors within a UA) are working together towards a common goal and one of them fails then the goal may fail too.

Furthermore, as we saw in the implementation Chapter 5, the reflection layer of eVATAR features a monitoring mechanism for the connections with the agent and avatar interactor software. Every connection in eVATAR is managed by a separate thread of program execution. If a connection is inactive, this thread will be unnecessarily retaining memory and processing resources with a potential impact on the scaling of the system (see 7.1). If the monitoring mechanism of the reflection layer of eVATAR detects failure in a connection, it will terminate the thread freeing all memory and processing resources (see 5.2.2).

7.2.2. Failure at eVATAR level & Continuity

There is a high dependency of the UAs on the functioning of the central hub i.e. the device that runs eVATAR. Failure of the central hub renders all UAs that are connected to it inoperative. Splitting the workload over multiple instances of the eVATAR middleware and when it is possible among distributed networked hosts would be a good way to ensure isolation of such failures into the subset of the UAs that are connected to the affected hubs. Furthermore, eVATAR has its own mechanisms for ensuring continuity of service.

As we saw in the implementation chapter 5 regarding the local failover, there is a backup instance of eVATAR running on the same host computer as the active instance of eVATAR. The backup eVATAR is monitoring the active instance of the middleware (see heartbeat in 5.1.2). In the case of failure of the active eVATAR, the monitor eVATAR will identify the failure (heartbeat reading

failure), assume the role of the active middleware using the same network address and start another monitoring redundant instance aiming to maintain service continuity. The interactors will have to reconnect and register again with the new instance of eVATAR.

The failover continuity mechanisms of eVATAR were tested on the hardware that was described in 7.1. Finally, hardware failure in the host machine that runs eVATAR in the currently supported setting would mean failure of the overall system. In the future work section we will discuss and propose a solution based on networked redundant nodes for running monitoring instances of eVATAR.

7.3. Comparing eVATAR against related work

In this section we compare eVATAR against related work in terms of accomplishing the aims of the thesis. According to the criteria set in section 2.1.2 of the “Background” chapter 2, we selected and reviewed middleware approaches that could potentially be used for integrating software agent intelligence with sensor and actuator networks (for the descriptions and reviews of each middleware see sections 2.2, 2.3 and 2.4). We have justified in section 2.5 and sections 2.2.4 and 2.3.6 why the middleware approaches of sections 2.2 and 2.3 respectively even though they achieve integrations of MAS AI to SAN they do not achieve all of the aims of the thesis simultaneously.

Furthermore, in section 2.5 we identified and justified that the approaches of 2.4 (high-level pervasive middleware) are the closest comparators to what we are trying to achieve. Furthermore each of these approaches was described and reviewed against the aims of the thesis in 2.4 and Table 2-2 provides with an overview of this. Table 7-2 extends this table with the inclusion of eVATAR. The first case study in section 6.1 provided with evidence that eVATAR satisfies all 5 aims of the thesis simultaneously and in section 6.1.3 we discuss how exactly it provides evidence for each aim.

Middleware	Aim 1 MAS-sensor/actuator/SAN integration capability	Aim 2 systematic integration of MAS with sensors/actuators/SAN	Aim 3 transparent integration of MAS with sensors/actuators/SAN	Aim 4 multiple MAS support	Aim 5 heterogeneity
eVATAR	yes	yes	yes	yes ¹	yes ²
SALSA	yes	yes	no	no	yes
Amigo	yes	no	yes	yes	yes
RoboCare	yes	no	yes	no	yes
iCore	yes	yes	no	no	yes
ReMMoc	no	no	no	no	yes
PEIS	no	no	no	no	yes
Middle Layer	yes	no	yes	no	no
MARIE	no	no	yes	yes	yes
Player	no	no	no	no	yes

¹ Provided proof of concept with GOLEM and JADE. We need to try more MAS platforms.

² Currently the API is restricted to JAVA platforms. See future work.

Table 7-2. Evaluation of pervasive/AmI middleware and eVATAR in relation to the aims of the thesis.

Table 7-3 (below) extends Table 2-1 by including eVATAR. It provides a cross-reference of approaches, technologies and characteristics of middleware that purport to support the integration of agent AI to sensor and actuator networks.

Middleware	Domain	Programming Paradigm	Interface	Heterogeneity	Reflection	Built-in intelligence	Inter-process Communication	Transparency
eVATAR	Pervasive, sensor/actor networks	SOA	declarative	yes	yes	No	asynchronous, MOM	high-level
SensorWare	WASN	database, agents	declarative	no	no	basic	asynchronous, tuples	low-level
Agilla	sensor/actor networks	agents	coding (imperative)	no	yes	basic	asynchronous, tuples	low-level
Impala	WSN	agents, event driven	-	no	yes	basic	asynchronous	no
SIXTH	WSN	component based - OSGi	declarative	yes	no	no	asynchronous	high-level
Sensor Web - NOSA	SN, WSN	SOA	declarative	no	no	no	asynchronous-OGC - broker- OOM	high-level
OASIS	SN, pervasive	SOA	declarative	yes	no	no	asynchronous	high-level
TinySOA	WSN	SOA	declarative	no	no	no	base station (broker - asynchronous)	high-level
USEME	sensor/actor Networks	SOA	declarative	no	no	no	synchronous & asynchronous	high-level
SALSA	AmI	agents, SOA	coding	yes	yes (adaptive)	basic	asynchronous, MOM-broker	high-level
Amigo	pervasive, AmI	SOA	declarative	yes	no	no	asynchronous, MOM-broker	high-level
RoboCare	pervasive, AmI	SOA, agents	declarative	yes	no	no	asynchronous, MOM-brokers	high-level
iCore	IoT	agents, SOA	declarative	yes	no	no	asynchronous, MOM-broker	high-level
ReMMoc	pervasive	pub-sub, web services	declarative	yes	yes	no	asynchronous, MOM - broker	high-level
PEIS	pervasive	pub-sub	coding	yes	no	no	tuples	high-level
Middle Layer	robotics, Pervasive	pub-sub	declarative	no	no	no	asynchronous, MOM - broker	high-level
MARIE	robotics, pervasive	mediator	declarative	Yes	no	no	asynchronous, broker	high-level
Player	pervasive	message queues	coding	Yes	no	no	asynchronous, MOM - broker	low-level

Table 7-3. Cross-reference of approaches and technologies used by the different middleware.

Having established that eVATAR is tailored to achieve the aims of the thesis, Table 7-3 is useful because it shows the fusion of technologies that were used to implement it. This combination of technologies and features could be used as a

starting point by developers that could potentially attempt the development of a similar middleware with similar aims. It can be contrasted to other middleware in Table 7-3 that implement different technological approaches.

Despite of providing with evidence for the aims of the thesis, we identify the following issues in the current implementation of eVATAR:

- Heterogeneity limitation due to the JAVA dependency of the eVATAR API. We should note that this is only a limitation of the current implementation of the eVATAR API. Any platform/programming language that supports TCP connectivity and basic XML parsing should be able to implement an API that can connect to and exchange messages with eVATAR (see future work plans in section 8.2). Furthermore, we have not tested eVATAR with all the platforms and technologies of 3.2.2.
- So far we have only tested the system using two MAS platforms, GOLEM [82] and JADE [25]. The application using JADE especially was simple and its purpose was to provide with a proof of concept. We need to test with more MAS platforms to strengthen the claim for multiple MAS support.
- As we saw in the beginning of this chapter, we have selected performance over scalability by implementing a multithreaded TCP server within eVATAR to manage the connections with the sensors and actuators. The question here is what we would need to do to improve scalability if an application requires it.
- eVATAR requires both metadata describing service providers as well as metadata describing service consumers in order to create internal models of avatars as well as of the agents. Consequently, as we have seen the service consumers register with eVATAR following the same process as the services by sending their metadata descriptions. We therefore implemented BIL tailored to a particular ontology that supports registration, discovery and interactions in eVATAR. The fact that it is tailored to a particular ontology allowed us to create compact BIL descriptions and messages as opposed to using more generic languages such as OWL (XML-based, consists of

knowledge representation languages for creating ontologies) [120] that need to cater for a variety of applications and ontologies. Nevertheless, the proprietary nature of BIL makes it less accessible to the research community. The SensorML (Sensor Model Language [123] and [124]) is a common standard for describing sensors and actuators using metadata and it could be investigated in terms of implementing the required by eVATAR ontology, making our API more accessible by the research community (see future work section 8.2.5).

In the future work section 8.2 we will describe how we will attempt to address the above issues.

Finally, by comparing eVATAR with other approaches in terms of achieving the aims of the thesis we satisfy objective VI of the thesis.

7.4. Summary - Conclusion

In this chapter we discussed eVATAR in terms of performance, points of failure and continuity in systems that use it. We have also seen how eVATAR compares in terms of achieving the aims of the thesis to the middleware approaches that were reviewed in chapter 2 and selected based on the criteria of section 2.1.2. We also identified a number of issues that eVATAR needs to improve on. In the following chapter this thesis concludes with a discussion about ideas for future work in terms of confronting these issues and a brief epilogue.

8. EPILOGUE

This last chapter concludes the thesis by presenting the synopsis of our work from the perspective of how it advances the state of the art. The chapter continues with a section presenting ideas for future work focusing on plans for improvements and further evaluation.

8.1. Synopsis

This section aims to clearly set out the novelty and advancement of the state of the art deriving from this thesis. There are numerous middleware approaches for integrating MAS AI to sensor (and potentially actuator) networks. In this thesis we are looking for a middleware approach with the following characteristics (which are also the aims of the thesis):

1. Capability to integrate software agent functionality with a SAN.
2. The framework should be systematic in the way it integrates software agent functionality with sensors and actuators.
3. The framework should be transparent to the system developers in the way it integrates software agent functionality with sensors and actuators.
4. The framework should support multiple MAS platforms.
5. The framework should support heterogeneous sensors, actuators and devices.

We reviewed numerous middleware approaches in the search for a middleware that simultaneously features all 5 characteristics. In particular we identified three categories of middleware that can integrate MAS AI with SAN: agent-based SAN / WSN middleware, SAN / WSN middleware that support integration with external resources such as a MAS and high-level middleware for pervasive systems. Having reviewed middleware from each category in sections 2.2, 2.3 and 2.4 we identified a paucity of middleware frameworks (see section 2.5) that satisfy all aims of the thesis simultaneously.

We therefore set a number of objectives (see section 1.3) for studying the problem and providing and evaluating a novel solution that would advance the state of the art with a middleware that satisfies all of the aims of the thesis.

We used the Z-notation [45] to specify a framework (chapter 4) for a middleware that fulfils the aims of the thesis as stated above (objective I). We implemented eVATAR based on the Z-Notation framework (chapter 5) satisfying objective II. To achieve objective III we then implemented a case study using eVATAR, the GOLEM MAS platform, the JADE MAS platform and sensors and actuators embedded in a miniature smart home (see chapter 6 and section 6.1). In this setting we implemented two scenarios, a security scenario using GOLEM and a simpler “light control” scenario using a JADE agent. These scenarios provided with evidence that eVATAR achieves the aims of the thesis. Thus we inferred that by using the Z-Notation framework of chapter 4 we specified the functionality of eVATAR, a middleware that fulfils the aims of the thesis. This way we also provided evidence for the Z-Notation framework.

To further test and evaluate our approach we created a test suit (see objective IV in section 1.3) consisting of a smart home simulation and GOLEM (section 6.2). We then implemented a scenario that shows in simulation the potential of our approach in providing useful services in our everyday lives (in our case saving on electricity consumption). This simulation was also used to measure latency in controlled environment tests (objective V) leading to useful insights regarding the implementation of eVATAR and in particular the trade-off between performance and scalability (section 7.1).

Finally, in section 7.3 we compared eVATAR to other approaches (objective VI) and summarized how we advance the state of the art by adding a novel middleware solution achieving all the aims of the thesis. We also provided with a comparative table (Table 7-3) showing the fusion of technologies that were used to implement eVATAR being contrasted to other systems in Table 7-3 that implement different technological approaches. This combination of technologies and features could be used as a starting point by developers that could potentially attempt the development of a similar middleware with similar

aims. We also identified a number of issues that will be confronted as part of our plans for future work.

8.2. Future Work

As future work, we intend to confront the issues identified in the thesis as well as to extend the scope of the connected devices that can be part of our architecture. It is also in our plans to further test and evaluate our approach. In the following we describe in more detail how we are going to achieve the above.

8.2.1. Performance vs Scalability

In chapter 7, we saw a discussion regarding the base layer server implementation of eVATAR that follows a multithreaded approach offering dedicated connections to DIS but also presenting scalability limitations. As future work, we intend to provide with a second implementation of the base layer of eVATAR that focuses on scalability using event driven technologies such as the one offered by the JAVA Apache “Mina” framework [101] or the one of “libuv” [111] (in this case we would either need to use a JAVA port of libuv or we would need to port eVATAR in C++). Following a modular approach, the intention is to enable system designers to use either of the two server implementations by editing a simple configuration file. Depending on the requirements of an application, the question that the system designers would need to ask would be whether a small overhead is acceptable in favour of scalability or whether performance is important on the expense of using fewer DIS (enough for most applications as we have seen) as in the current implementation. The rest of the functionality of the middleware should remain unaffected by this configuration.

8.2.2. Continuity of Service

Furthermore, in 7.2.2 we identified a concern stemming from the use of a broker in eVATAR and specifically from the fact that a hardware failure in the host machine that runs eVATAR would mean failure of the overall system. Hardware problems in host machines are quite common in server applications. We intend to investigate the potential of using architectures that utilize High-Availability (HA) clusters [22]. HA clusters are groups comprising of instances of eVATAR running on the same or on distributed hosts (computers, tablets etc.). The availability of distributed networked hosts allows for the deployment of remote failover mechanisms to further ensure continuity of service. The concept for the remote failover is similar to the local one that was described in 5.1.2 with a key difference. In this case the heartbeat is implemented as a request-reply protocol over a TCP connection [49]. The idea is that in the case of hardware failure, the distributed interactor software will reconnect to a different instance of eVATAR that runs on a different hardware host.

8.2.3. Expanding the Scope of Supported Devices

In addition to the above, we intend to use different technologies for the eVATAR API that is currently implemented as a JAVA Jar. The JAVA version of the API is well suited for platforms that support Java. Despite this, there is a need for support of platforms that are not running the JAVA virtual machine (JVM) and also a need for addressing the concerns regarding the memory and processing power overhead of installing JAVA on software that control distributed sensors and actuators. This leads to a requirement for an alternative implementation of the eVATAR API as a dynamic library. Any platform/programming language that supports TCP connectivity and basic XML parsing should be able to implement an API that can connect to and exchange messages with eVATAR

This implementation will be using C++ to follow an object oriented design that is similar to the one that was presented above for the JAVA version of the

eVATAR API. Two types of dynamic libraries should be implemented: a “shared object” for Linux based systems [100] and a “dynamic-link library” for the Microsoft Windows operating system [114]. The C++ implementation will be exposed by the dynamic libraries to calling applications (the distributed interactor software) using a low-level C based interface in order to further ensure portability. In terms of functionality, all versions of the API should be identical and it is up to the system developer to decide which one would be better suited for a particular implementation setting.

PUCK [117] offers an interesting approach for an alternative way of integrating devices with MAS AI via eVATAR which can expand the scope of eVATAR with regards to the platforms it can use. PUCK is part of OGC-SWE framework (OGC Sensor Web Enablement standard) and it is a simple standard command protocol that automates the process of installing, configuring and operating devices by physically storing information about a device within the device itself.

It features a simple protocol that defines a small “datasheet” that can be retrieved from devices that support PUCK and contains identifying metadata. It also stores additional information in the “payload” that is also accessible from the device. The payload can contain any type of information. Using a BIL description as a PUCK payload allows us to describe an instrument’s command protocol in a standard way (a similar approach has been followed by using SensorML [123] payloads [117]). This way we describe such instruments as service providers. By doing this, eVATAR should in principle be able connect to any PUCK enabled instrument that is described by this file. In terms of connectivity between the devices and eVATAR, PUCK allows connectivity over RS-232 interface or over TCP/IP using an Ethernet cable. As part of our future work plans we are going to adapt eVATAR to accept such connections over TCP/IP and this way enable agents to control the PUCK compliant instruments via eVATAR. Thus the scope of the devices that can use eVATAR is extended to devices that are PUCK compliant. By doing this we enable agent AI to be

integrated via eVATAR to a whole new range of applications in the field of maritime observations.

8.2.4. Deployment - Usability

The limited availability of hardware and testing environments during the writing of the thesis was a problem we had to confront. Whilst the case studies and the performance testing proved quite successful in terms of allowing us to review the proposed framework for UAs and eVATAR using real sensors/actuators and a simulation, it would be important to evaluate eVATAR in an even more realistic world scenario deployment e.g. in a smart home with a human inhabitant, in a museum or in a smart city context. Furthermore, as part of our future work plans we intend to use eVATAR with as many technologies of 3.2.2 as possible to identify potential issues and improve eVATAR in terms of heterogeneity support.

The question around usability investigates the steepness of the learning curve of the middleware. It is not uncommon in modern middleware to see such level of complexity where only a few of experts have the experience and skills to use them in terms of application design. The philosophy behind eVATAR strives for the exact opposite. To remove complexity from the task of integrating MAS with sensor networks and therefore it strives for a simple to use API. Poorly designed or complex APIs would negate the whole purpose of using eVATAR. As a part of the future work with eVATAR, we anticipate that eVATAR will be evaluated in terms of usability by system designers that will use it in their applications.

8.2.5. From BIL to a SensorML ontology

SensorML (Sensor Model Language [123], [124]) is an international standard and it is also part of the Open Geospatial Consortium (OGC) Sensor Web Enablement (SWE) [35] standards. It is an extensible Markup Language (XML) representation used to describe different aspects of sensor (and actuator)

systems including process model, process chain, connections and system physical layout among others (for more details see [123]). The Component element of SensorML is used for transforming information from one format to another. The ProcessModel element is a simple atomic process providing a functional model of a sensor, an actuator or a filter using metadata. It is defined using the ProcessMethod element that defines an interface to a process as well as its inputs, outputs and parameters. The ProcessChain element is a composite processing block consisting of interconnected process models or other process chains.

SensorML supports the implementation of different ontologies. Building on the above elements we can build an ontology similar to the one of BIL. In particular we can create avatar/agent process chains consisting of process model sensors and actuators. Therefore, as part of our future work plans we will investigate ways to create the ontology of BIL using SensorML to standardize the process and make it more accessible to the research community.

8.2.6. Security

A general requirement for middleware that are applied to AmI scenarios, including eVATAR, deals with security issues. Numerous novel ways for shifting sensitive data within distributed AmI environments in the context of integration, collaboration and communication have been proposed and this data needs to be protected. Applications using these middleware aspire in principle to become integral to the everyday life of people. If they were compromised by a malicious source, they could pose a physical danger to the same people whose activities they were designed to support. Researchers need to address the security issues and this is one of the main concerns that will inspire future research with eVATAR. As we have seen eVATAR enables localized applications (in local area networks) with the MAS being the single point of communication with the outside world. An extra layer of security in the MAS would allow the system to communicate with the internet and become part of larger architectures without allowing direct access to local sensors and

actuators and hence compromising their security (as also mentioned above in the discussion about using eVATAR in an IoT context).

8.2.7. Enabling Robots To Access Virtual Reality

So far we have seen connections in which software agents control avatars. The idea here is to investigate the potential of using eVATAR also for:

- a) robots using software/virtual agents and
- b) in the same application, robots controlling software agents as well as agents controlling robotic bodies or sets of sensors and actuators in the physical world (being the avatars as described in the approach of this thesis).

What is important here is which entity has the control and which entity or set of sensors and actuators constitutes the avatar body.

APPENDIX A

A-1

Describing sets that represent aspects of the middleware:

- **INTERACTOR_IDENTIFIERS:** a set of identifiers for the agent/avatar interactors. They are used in order to register agent and avatar interactors in the system.
- **ENTITY_IDENTIFIERS:** The set of unique identifiers for agents and avatars.
- **SERVICE_DESCRIPTIONS:** the set of avatar and agent interactor descriptions that describe them as service providers or service consumers. They refer to metadata registration descriptions that are stored as part of the metadata of interactors in the interactor metadata registration list (see section 3.3).
- **BINDING_TYPE:** the “BINDING_TYPE” set provides values that instruct the middleware on how to bind agents with avatars. Targeted binding connects agent interactors to specific avatar interactors while “agnostic” binding allows the middleware to decide which agent interactors will connect to which avatar ones.
- **TARGET:** the set of all target identifiers that are used if the binding_type is “targeted”. Not applicable for agnostic binding and the value would be -1.
- **ENTITY_TYPE:** declares the entity type which can be an agent or an avatar.
- **INTERACTOR_TYPE:** declares the interactor type which can be a sensor or an actuator.
- **SERVICE_TYPE:** declares whether the interactor presents service provider or a service consumer functionality.

- **MESSAGES:** the set of messages that are received by the middleware from the agent/avatar actuators and sensors. They refer to the messages created using meta-language as required by 3.4.1.
- **OUTCOME:** the set containing exactly two values, “ok” and “fail”. They are used in the specification to represent successful or failed operations and actions.

A-2

Describing the basic sets that represent the main aspects of the API:

- **INTERACTOR_IDENTIFIERS:** a set for all the unique interactor identifiers in the system.
- **ENTITY_IDENTIFIERS:** the set of unique identifiers for agents and avatars.
- **METADATA:** a set of metadata descriptions. A metadata description is a collection of data elements that describe agents, avatars and their interactors.
- **SERVICE_DESCRIPTIONS:** metadata descriptions that describe the services that interactors offer or consume.
- **MESSAGES:** messages to and from the middleware.
- **DIS_INPUT_DATA:** a set of values that describe interactor activity at runtime. They are passed to the API by the DIS when it uses the API to send a message to the DIS that is bound to it (via the middleware). For example physical sensors send sensory data request messages, agent sensors receive sensory data messages, agent actuators send action request messages and physical actuators receive action request messages.
- **INTERACTOR_TYPE:** set with values for describing the type of the interactor: {“sensor”, “actuator”}
- **BINDING_TYPE:** set with elements used for instructing the middleware on how to bind entities from one system to entities in another system. Targeted

binding connects interactors in one system to specific interactors in the other system as selected by the system designer. The “agnostic” binding allows the middleware to decide the binding of the interactors autonomously and without user involvement. Set values: {“targeted”, “agnostic”}

- TARGET: if the binding_type is “targeted” it provides the middleware with a target identifier value. Not applicable for agnostic binding, the value would be -1.
- ENTITY_TYPE: the entity type set which can have the elements: {“agent”, “avatar”}.
- SERVICE_TYPE: the service type set which can have the elements: {“provider”, “consumer”}.
- OUTCOME: the set containing exactly two values, “ok” and “fail” that will be used in the specification to represent successful or failed operations/actions.

A-3

The basic functions of the middleware API are described in the following table.

- idToMetaData: adding the identifier of the interactor to metadata.
- typeToMetaData: adding the type of the interactor (sensor/actuator) to metadata.
- bindingTypeToMetaData: adding the binding type of the interactor (targeted/agnostic) to metadata.
- targetToMetaData: adding the target value of the binding to metadata.
- entityIdToMetaData: adding the id of the entity that the interactor is a member of to metadata.

- `entityTypeToMetaData`: adding the type of the entity (agent/avatar) that bares the particular interactor to metadata.
- `bodyIdsToMetaData`: adding the set consisting of all interactor identifiers of the entity (agent/avatar) that bares the current interactor to metadata.
- `serviceTypeToMetaData`: adding the service type of the interactor (provider/consumer) to metadata.
- `serviceDescriptionToMetaData`: adding the service description data representation to metadata. Used in the registration description messages.

APPENDIX B

B-1

BIL uses the standard namespace. The URI associated with this namespace is the Schema language definition with the standard value “<http://www.w3.org/2001/XMLSchema>”.

Schema 1: XML Schema Definition (XSD) for BIL descriptions

XML Schema Definitions for BIL interactor description, as acquired from the bilDescription.xsd:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="bilDescription">
  <xs:complexType>
    <xs:sequence>

<xs:element name="bilinfo">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="interactorType" type="xs:string"/>
      <xs:element name="entityType" type="xs:string"/>
      <xs:element name="entityID" type="xs:string"
minOccurs="0"/>
      <xs:element name="body">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="sensor" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            <xs:element name="actuator" type="xs:string" minOccurs="0"
maxOccurs="unbounded">
              </xs:sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="communicationType" type="xs:string"/>
          <xs:element name="bindingType" type="xs:string"/>
          <xs:element name="target" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

<xs:element name="Service">
```

```

        <xs:complexType>
            <xs:sequence>
<xs:element name="ServiceType" type="xs:string"/>
<xs:element name="Attribute" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    </xs:sequence>
    <xs:attribute name="InteractorID" type="xs:string"
use="required"/>
    </xs:complexType>
</xs:element>
</xs:schema>

```

Figure B-0-1. BIL interactor description XSD.

We see in the above figure:

```
<xs:element name="entityID" type="xs:string" minOccurs="0"/>
```

The `minOccurs="0"` means that this element is optional. In general eVATAR does not necessarily need the bilentity xml documents for avatars. Especially in the case of smart homes consisting of independent devices, sensors and actuators all of them may register to eVATAR without declaring a relationship to an avatar body. eVATAR will group them into an avatar body and assign them an avatar identifier.

Schema 2: XML Schema Definition for BIL messages

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="bilmessage">
    <xs:complexType>
        <xs:sequence>

            <xs:element name="entityType" type="xs:string"/>
<xs:element name="entityid" type="xs:string"/>
<xs:element name="InteractorID" type="xs:string"/>

<xs:element name="Service">
            <xs:complexType>
                <xs:sequence>
<xs:element name="ValueInt" type="xs:integer" minOccurs="0"/>
                <xs:element name="ValueLong" type="xs:long" minOccurs="0"/>
                <xs:element name="TEXT" type="xs:string" minOccurs="0"/>
<xs:element name="XML" type="xs:string" minOccurs="0"/>
                <xs:element name="StatusOn" type="xs:string"
minOccurs="0"/>

```

```

        <xs:element name="Speed" type="xs:string" minOccurs="0"/>
        <xs:element name="Temperature" type="xs:string"
minOccurs="0"/>
        <xs:element name="Duration" type="xs:string" minOccurs="0"/>
        <xs:element name="Height" type="xs:string" minOccurs="0"/>
        <xs:element name="Length" type="xs:string" minOccurs="0"/>
        <xs:element name="ASSERT" type="xs:string" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>

</xs:sequence>
<xs:attribute name="messageID" type="xs:string"
use="required"/>
</xs:complexType>
</xs:element>
</xs:schema>

```

Figure B-0-2. BIL message XSD.

B-2

Child Elements of “Service”

Element	Description
ValueInt	it takes an integer value. It is typically used to describe a sensory data or an actuator control command as an integer value.
ValueLong	it takes a long integer value. It is typically used to describe a sensory data item or an actuator control command as an integer value.
TEXT	it takes a character string value. It is typically used to describe a sensory data item or an actuator control command as a string. There are no constraint on the contents of the string.
XML	it takes a string as a value. Acceptable strings are other structured xml documents.
StatusOn	it takes two strings: “ON” and “OFF”. It is typically used

	to set switch actuators on and off.
Speed	it takes a string as a value. Acceptable strings are: "FAST", "SLOW", "MODERATESPEED".
Temperature	sensor, it takes a string as a value. Acceptable strings are: "HOT", "COLD", "MODERATETEMP".
Duration	actuator, it takes a string as a value. Acceptable strings are: "LONGDURATION", "SHORTDURATION", "MODERATEDURATION".
Height	sensor, it takes a string as a value. Acceptable strings are: "HIGH", "LOW", "AVERAGEHEIGHT".
Length	it takes a string as a value. Acceptable strings are: "LONG", "SHORT", "MODERATELENGTH".
ASSERT	it takes a string as a value. Acceptable strings are: "YES" and "NO"
DATA	Used for fine grain communication. Signifies that we will not be using the BIL XML command but instead we will be sending data types in chunks of bytes. "DATA" takes as an input a text in XML format with information regarding the data that will be sent including data_type (e.g. integer values), name and size.
DATA_COMMAND	Used for fine grain communication in conjunction with "DATA". The DATA_COMMAND is used to signify the beginning and the end of the fine grain transmission. It takes string values: "START", "END". It is sent right after the DATA message. Sending data in bytes will speed up the interaction and its more suitable for fine grain communication.

Table B-0-1. Describing the child elements of the "Service" tag for the BIL message.

B-3

eVATAR Base Layer UML classes

In this section we see the detailed UML classes that correspond to Diagram 5-3.

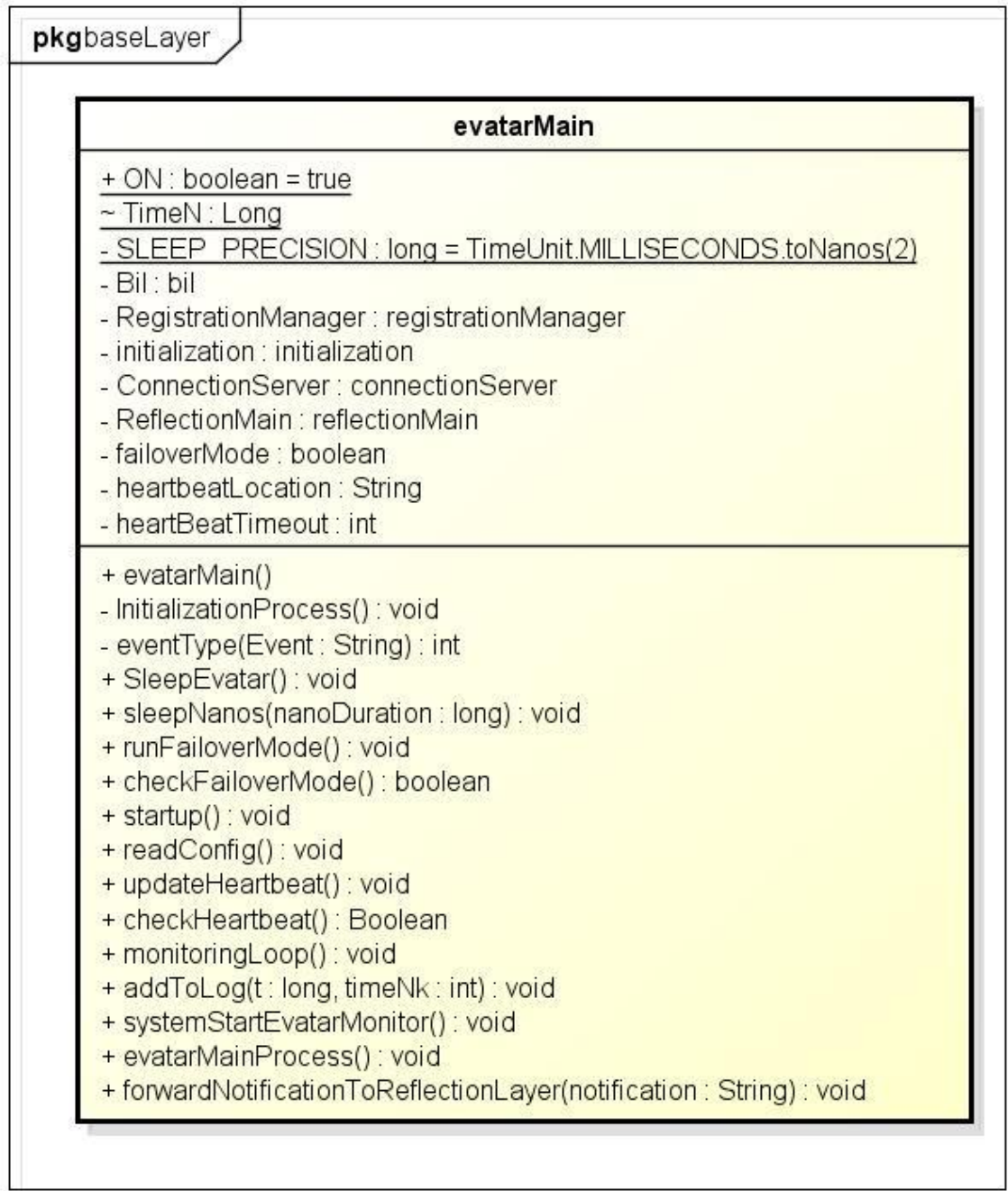


Diagram B-1. The "evatarMain" class.

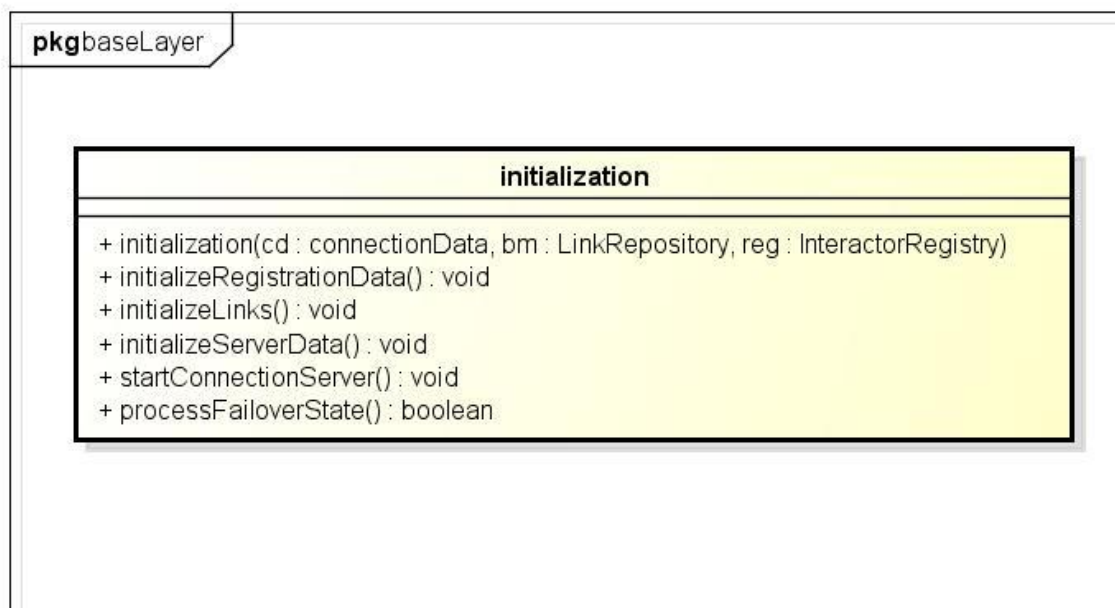


Diagram B-2. The "initialization" class.

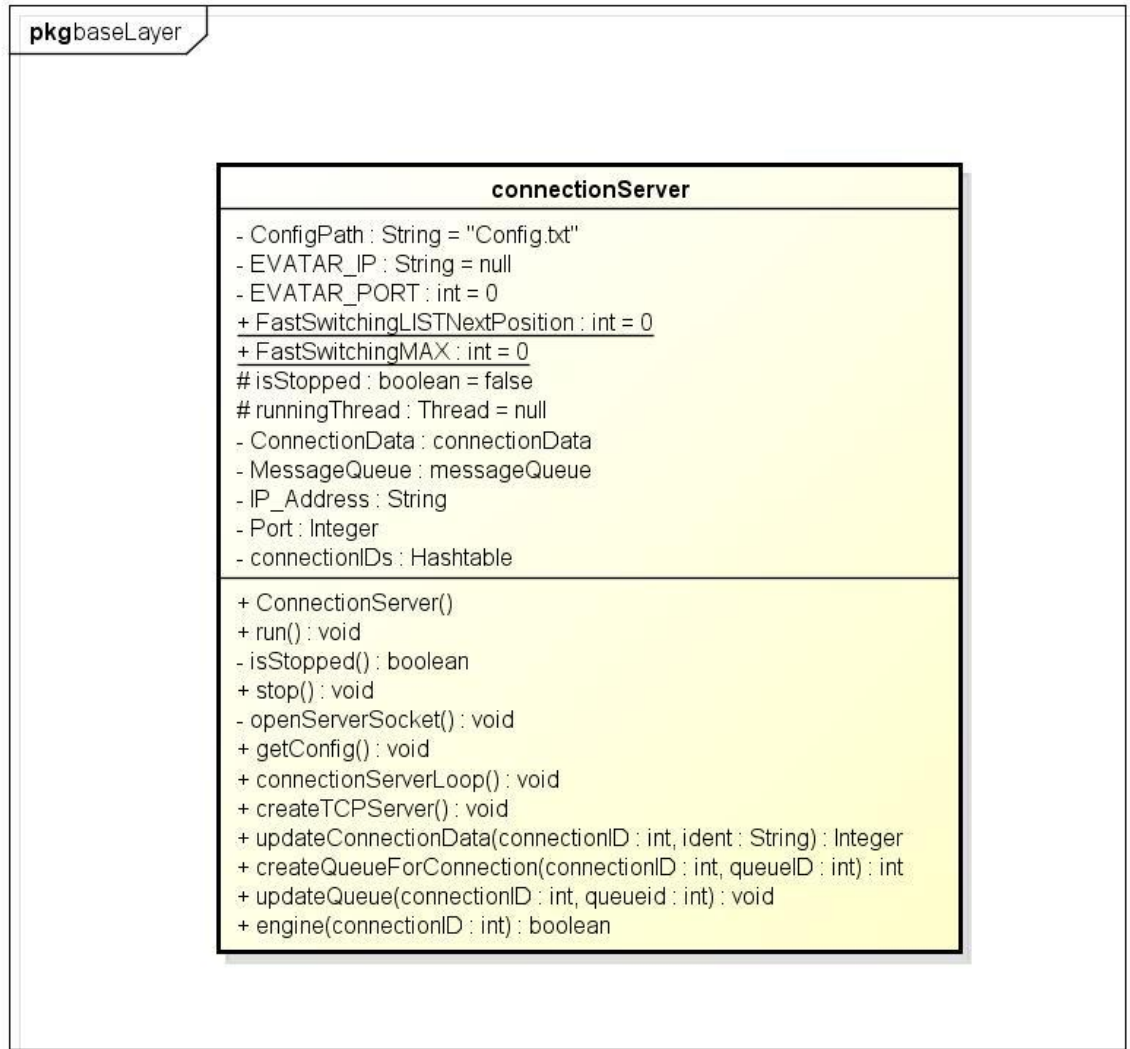


Diagram B-3. The "connectionServer" class.

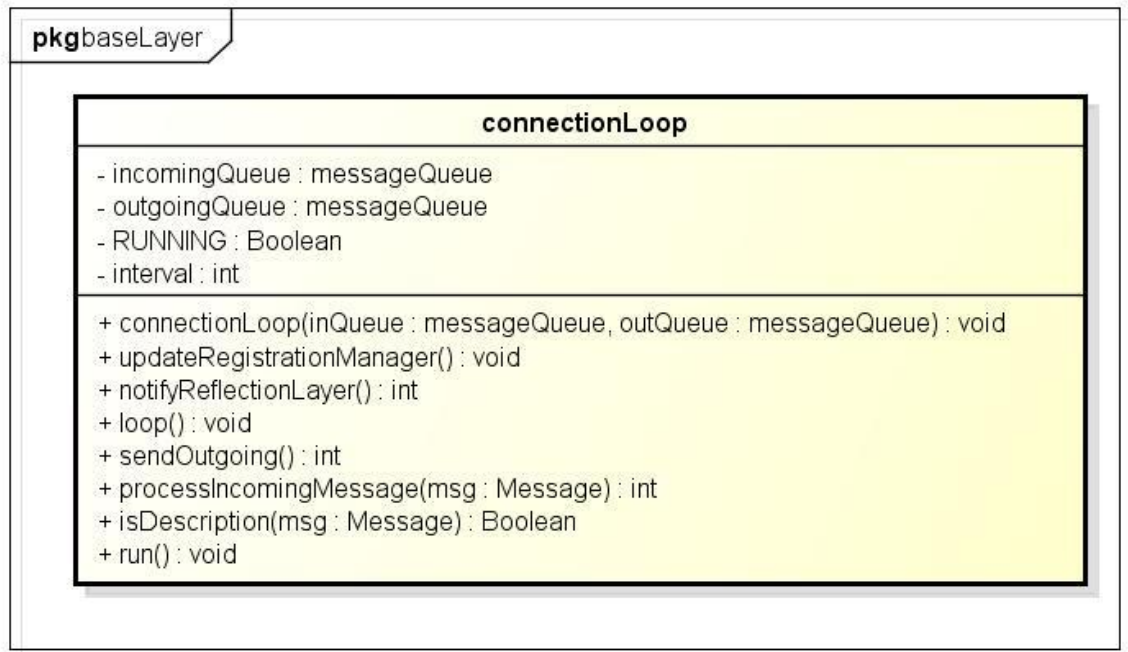


Diagram B-4. The “connectionLoop” class.

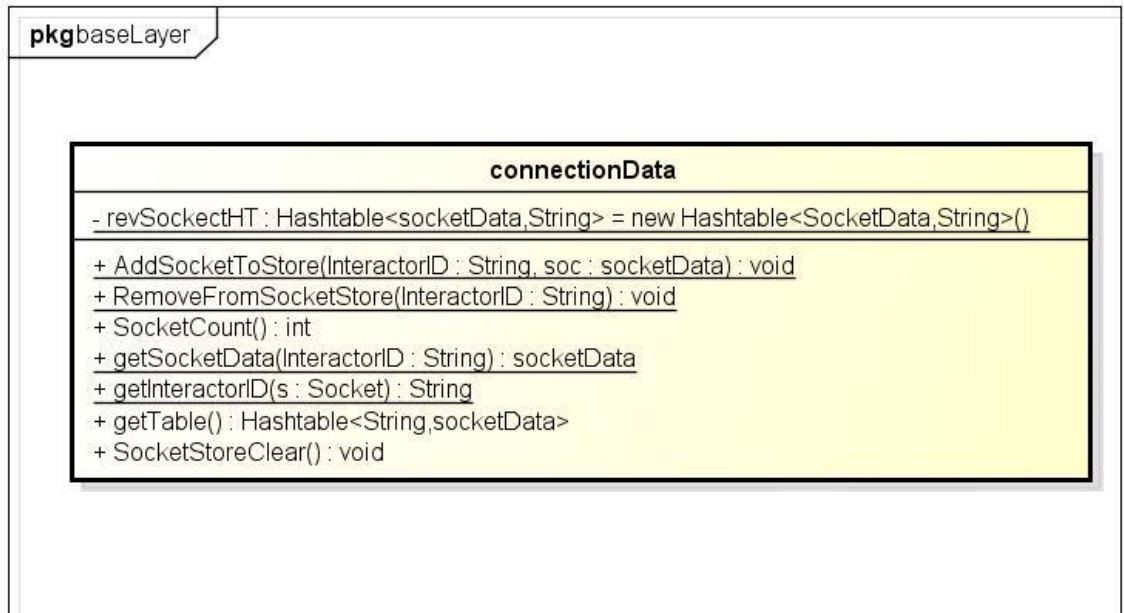


Diagram B-5. The “connectionData” class.

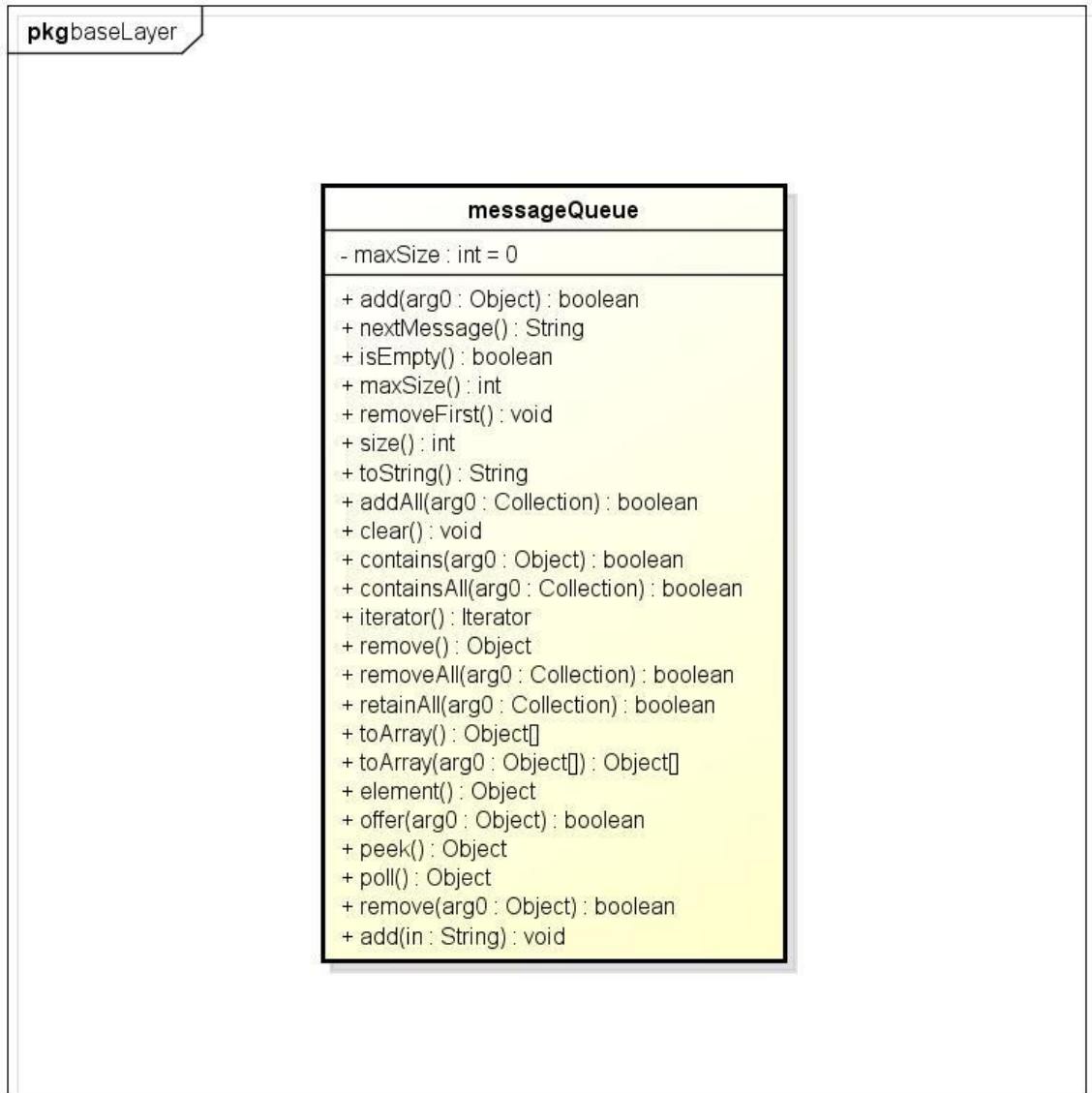


Diagram B-6. The “messageQueue” class.



Diagram B-7. The “registrationManager” class.

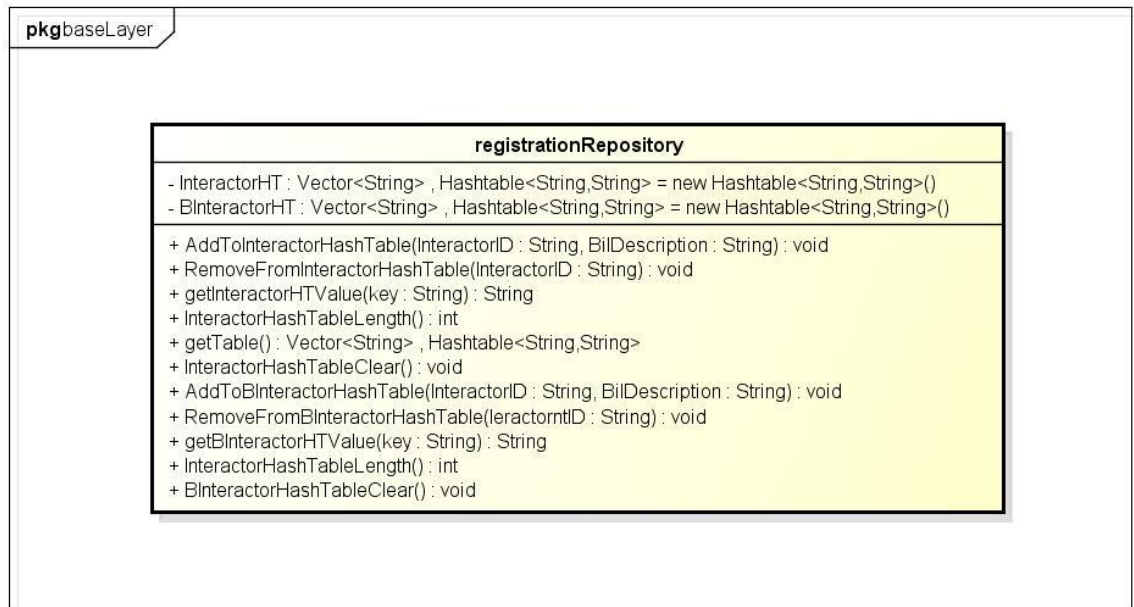


Diagram B-8. The “registrationRepository” class.



Diagram B-9. The “bil” class.

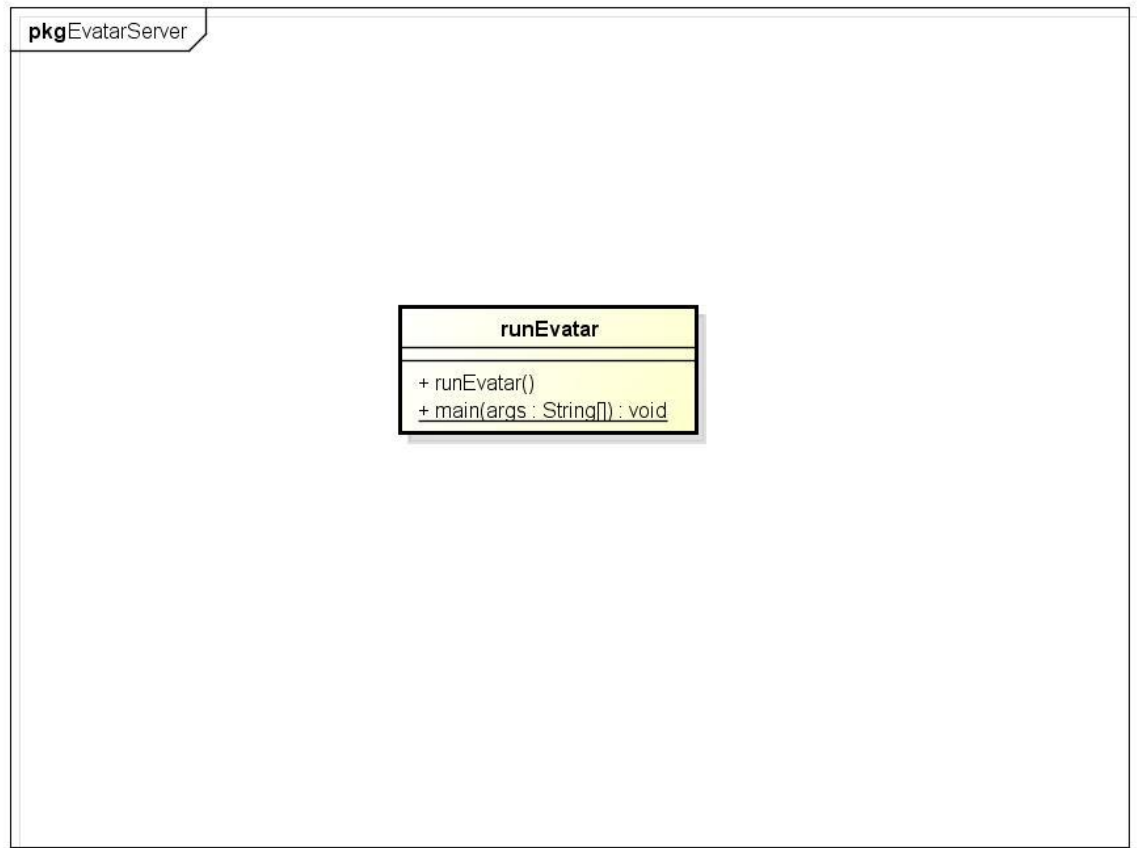


Diagram B-10. The “runEVATAR” class.

eVATAR API UML classes

In this section we see the detailed UML classes that correspond to Diagram 5-4.

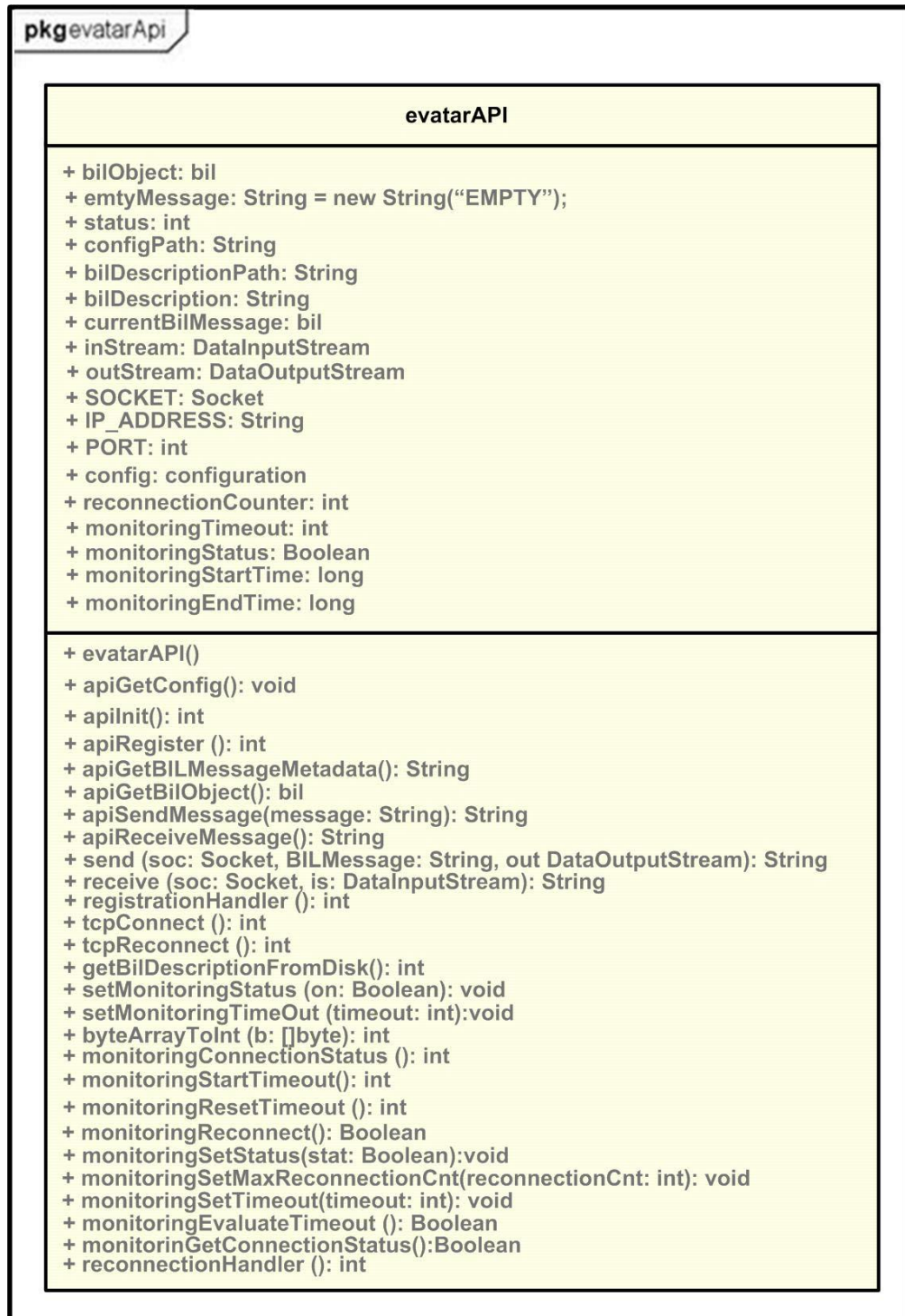


Diagram B-11. The "evatarAPI" class.

bil

```

~ currentSenderID : String
~ currentBilType : int = 0
~ currentEntityType : String
~ currentCommsType : String
~ currentSenderInteractorID : String
~ currentSenderIDFromBILMessage : String
~ ids : Vector<String> = new Vector<String>()
~ attrs : Vector<String> = new Vector<String>()
- BindingType : String
- Target : int
- ServiceType : String
- Description : String
- BodyIdentifiers : Vector<String>
- InteractorID : String
- EntityID : String
- InteractorType : String
- EntityType : String

+ BIL()
+ getSenderID(BILMessage : String) : String
+ getBILType(BILMessage : String) : int
+ getEntityType(BILMessage : String) : String
+ getCommunicationType(BILMessage : String) : String
+ getSenderInteractorID(BILMessage : String) : String
+ getSenderIDFromBILMsg(BILMessage : String) : String
+ getInteractorIDsFromBILDescription(BilDescription : String) : String[]
+ getAttributesFromBILDescription(InteractorBilDescription : String) : String[]
+ getValueIntAttributeValueFromBILMessage(BILMessage : String) : int
+ getValueLongAttributeValueFromBILMessage(BILMessage : String) : long
+ getTEXTAttributeValueFromBILMessage(BILMessage : String) : String
+ getNextTEXTAttributeValueFromBILMessage(BILMessage : String, next : int) : String
+ getXMLAttributeValueFromBILMessage(BILMessage : String) : String
+ getStatusOnOFFAttributeValueFromBILMessage(BILMessage : String) : String
+ getSpeedAttributeValueFromBILMessage(BILMessage : String) : String
+ getTemperatureAttributeValueFromBILMessage(BILMessage : String) : String
+ getDurationAttributeValueFromBILMessage(BILMessage : String) : String
+ getHeightAttributeValueFromBILMessage(BILMessage : String) : String
+ getLengthAttributeValueFromBILMessage(BILMessage : String) : String
+ getYESNOAttributeValueFromBILMessage(BILMessage : String) : String
+ setValueIntAttributeInBILMessage(BILMessage : String, value : int) : String
+ setValueLongAttributeInBILMessage(BILMessage : String, value : long) : String
+ setTEXTAttributeInBILMessage(BILMessage : String, val : String) : String
+ setNextTEXTAttributeInBILMessage(BILMessage : String, value : String, index : int) : String
+ setXMLAttributeInBILMessage(BILMessage : String, val : String) : String
+ setStatusOnOFFAttributeInBILMessage(BILMessage : String, val : String) : String
+ setSpeedAttributeInBILMessage(BILMessage : String, val : String) : String
+ setTemperatureAttributeInBILMessage(BILMessage : String, val : String) : String
+ setDurationAttributeInBILMessage(BILMessage : String, val : String) : String
+ setHeightAttributeInBILMessage(BILMessage : String, val : String) : String
+ setLengthAttributeInBILMessage(BILMessage : String, val : String) : String
+ setYESNOAttributeInBILMessage(BILMessage : String, val : String) : String
+ readBILtoString(BilXMLPath : String) : String
+ createEventDescription(data : String) : String
+ setBindingType(bindingType : String) : void
+ SetTarget(target : String) : String

```

Diagram B-12. The “bil” class.

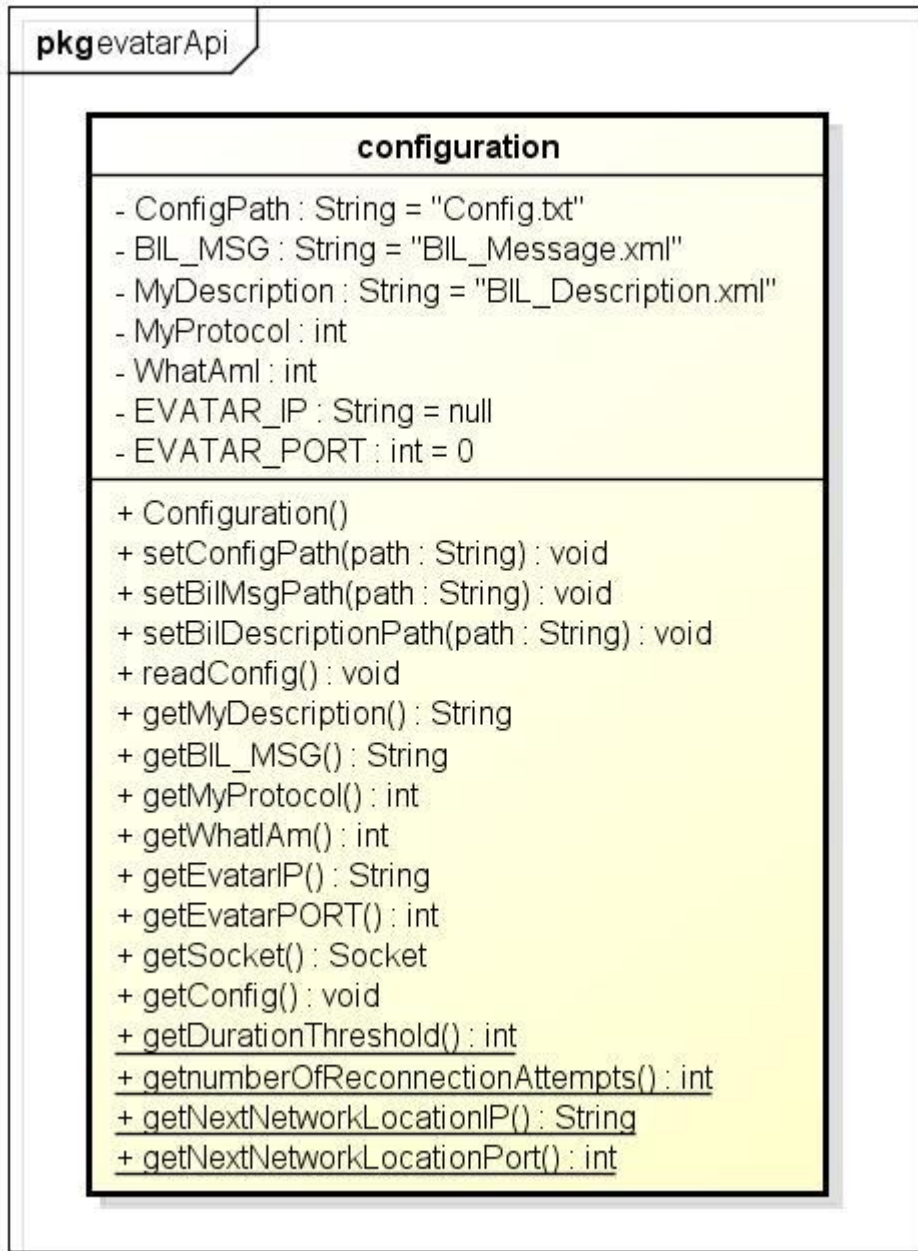


Diagram B-13. The “configuration” class.

Models Package UML classes

In this section we see the detailed UML classes that correspond to Diagram 5-5.

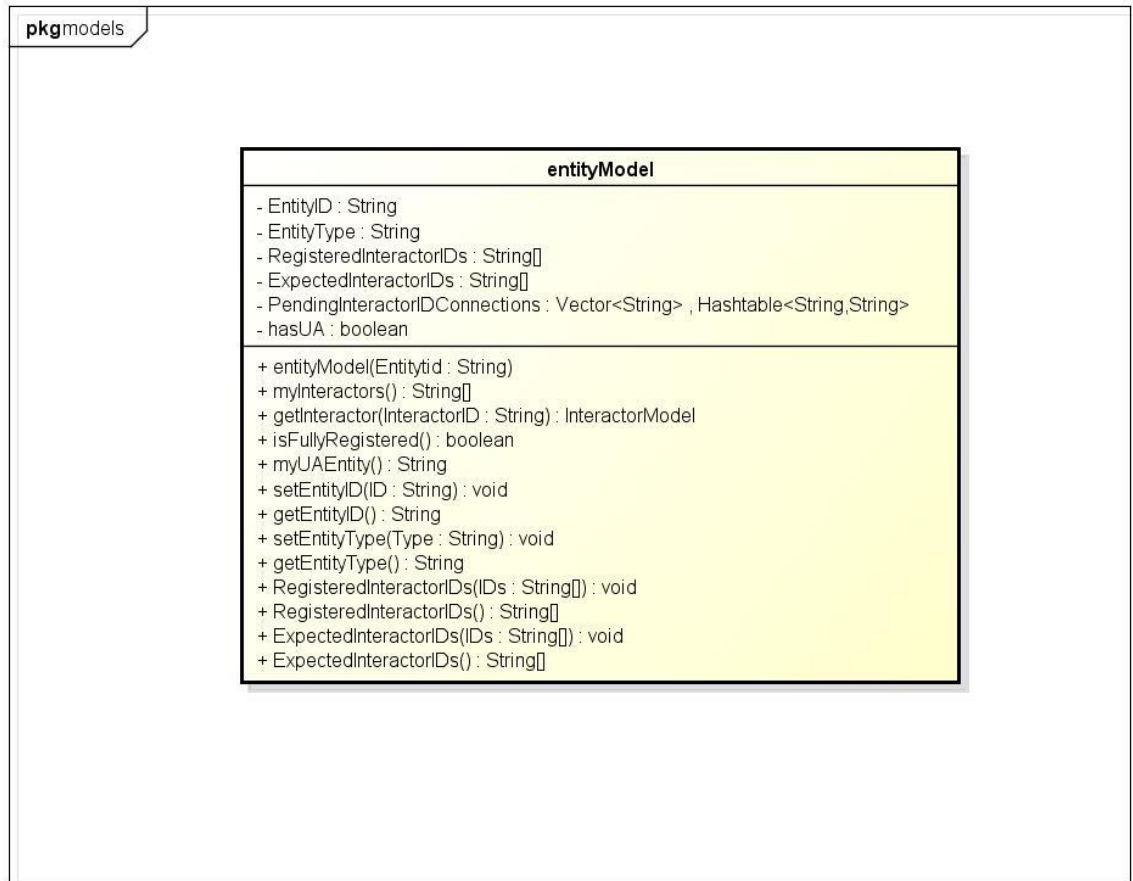


Diagram B-14. The “entityModel” class.

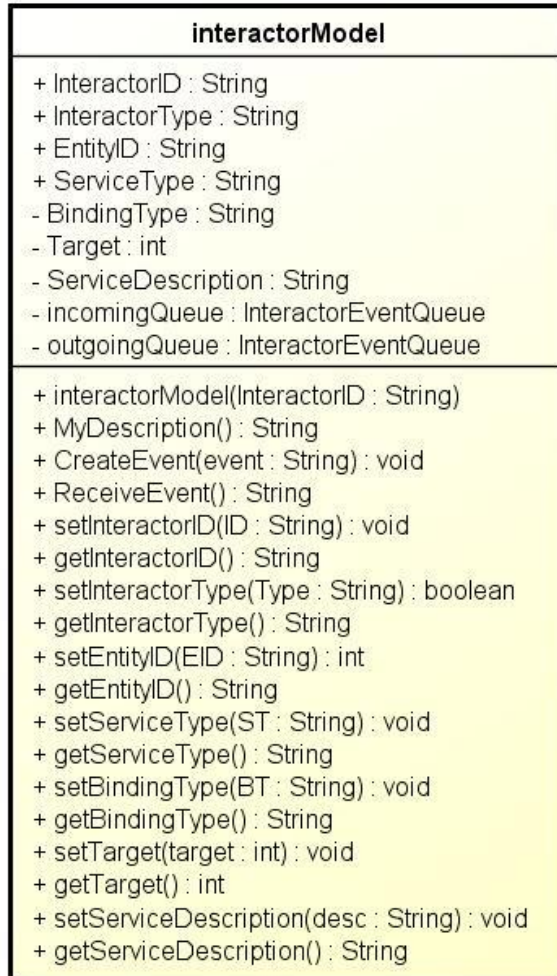


Diagram B-15. The “interactorModel” class.

Reflection functionality package UML classes

In this section we see the detailed UML classes that correspond to Diagram 5-6.

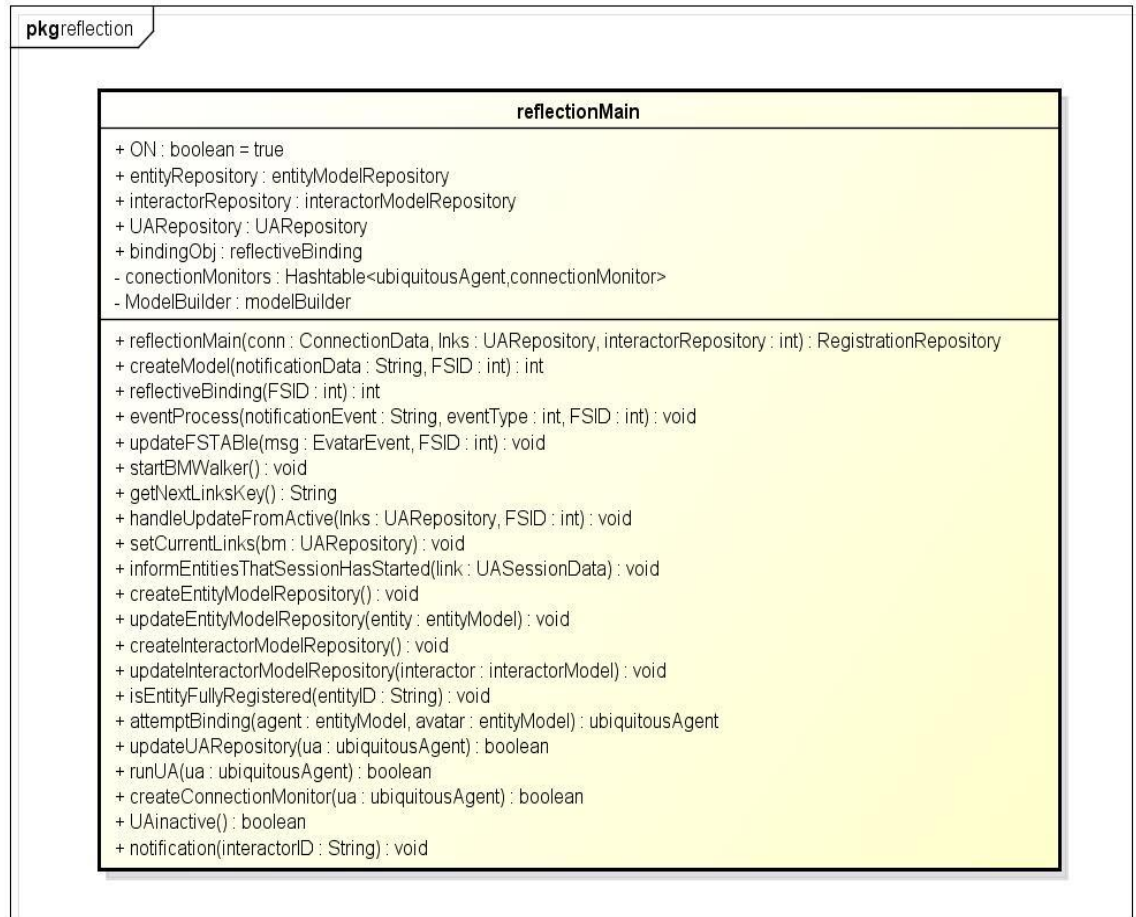


Diagram B-16. The “reflectionMain” class.

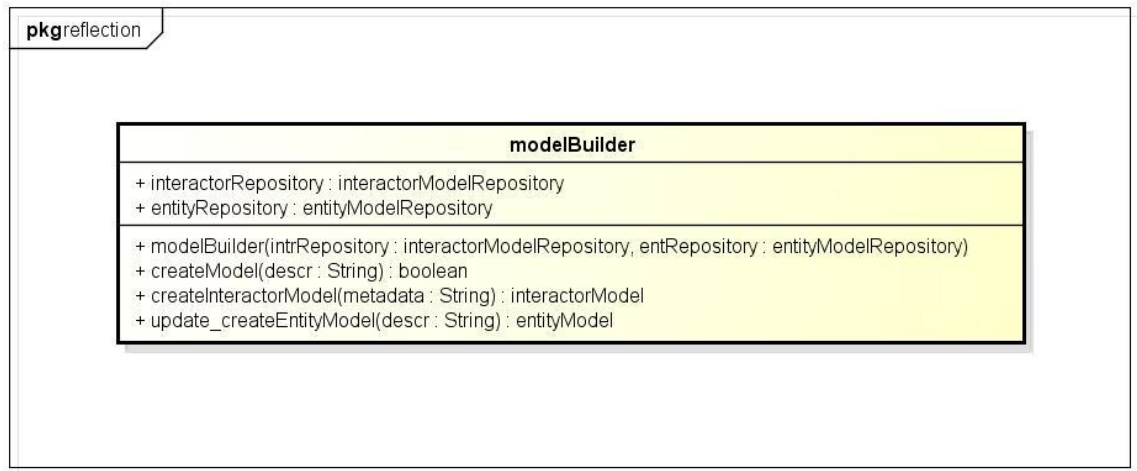


Diagram B-17. The “modelBuilder” class.

pkg reflection

entityModelRepository

- EntityHTB : Hashtable<String,entityModel>
- numberOfEntities : int = 0

+ entityModelRepository()
+ addEntity(entityModel : entityModel) : void
+ removeEntity(EntityID : String) : void
+ registeredEntitiesCount() : int
+ getEntity(entityID : String) : entityModel
- startEntityTableWalker() : boolean
- getNextEntityHTKey() : String
+ registeredEntities() : entityModel[]

Diagram B-18. The "entityModelRepository" class.

pkg reflection

InteractorModelRepository

```
- InteractorHTB : Hashtable<String,interactorModel>
- numberOfInteractors : int = 0

+ interactorModelRepository()
+ addInteractor(interactorModel : interactorModel) : void
+ removeInteractor(interactorID : String) : void
+ registeredInteractorsCount() : int
+ getInteractor(interactorID : String) : interactorModel
- startInteractorTableWalker() : boolean
- getNextInteractorHTKey() : String
+ registeredInteractors() : interactorModel[]
```

Diagram B-19. The “interactorModelRepository”.

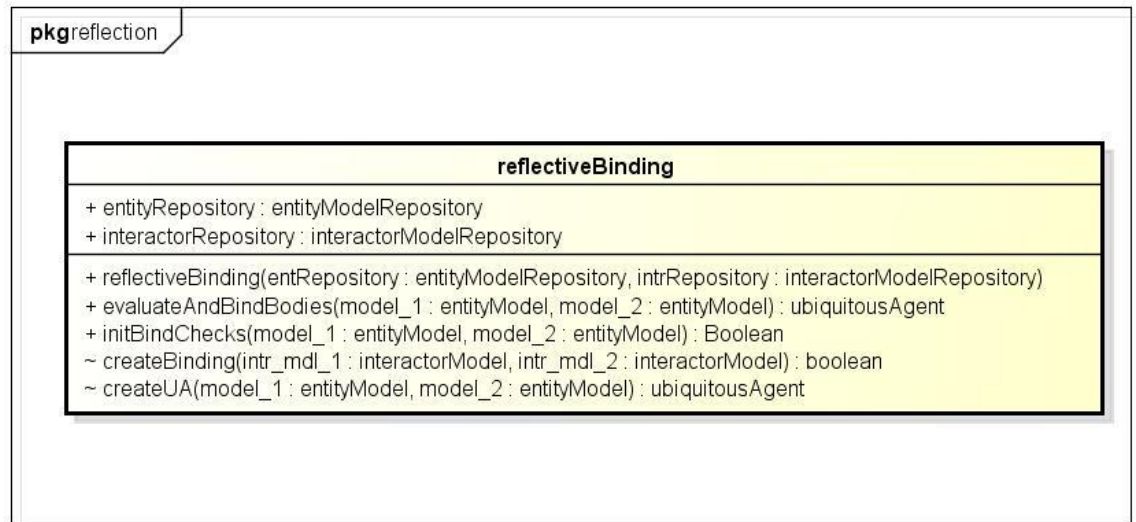


Diagram B-20. The “reflectiveBinding” class.

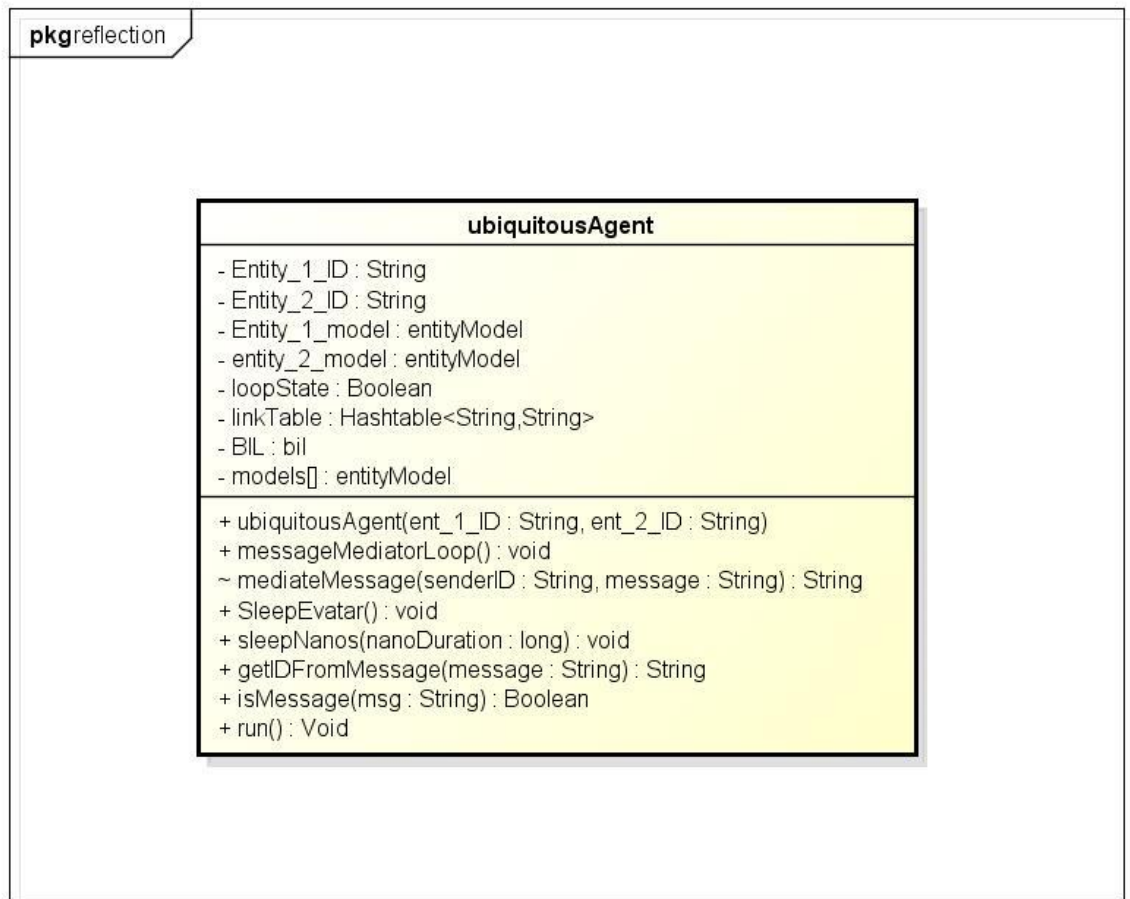


Diagram B-21. The “ubiquitousAgent” class.

pkg reflection

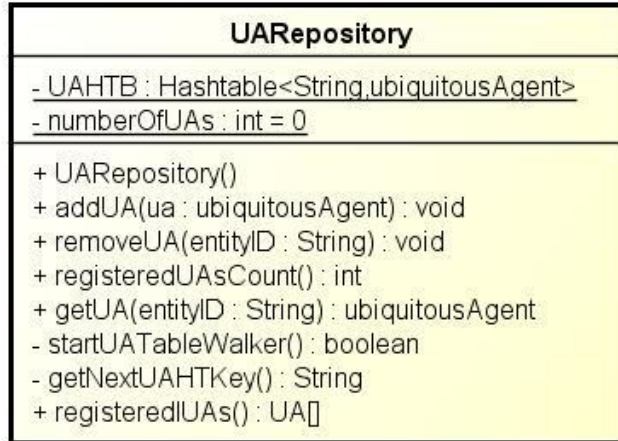


Diagram B-22. The “UARepository” class.

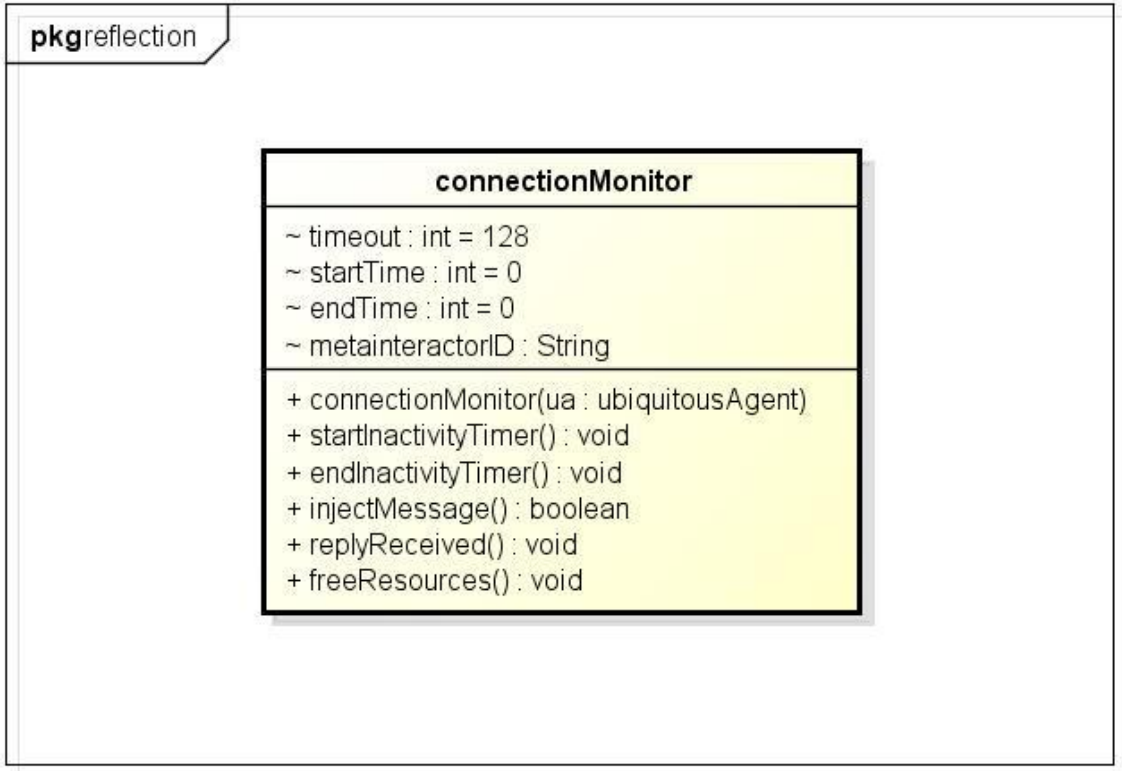


Diagram B-23. The “connectionMonitor” class.

APPENDIX C

C-1

RoboHome Avatars:

1st Avatar - Electricity consumption monitoring:

Sensors	<ul style="list-style-type: none">• 6 Electricity consumption Sensors from Light in 6 rooms (kitchen, living room, bedroom, corridor, toilet and laundry).
Total 11	<ul style="list-style-type: none">• Electricity consumption of TV (in use and standby).• Electricity consumption of Sound System (in use and standby).• Electricity consumption of Blu ray-media player (in use and standby).• Electricity consumption of washing machine.• Electricity consumption of dishwasher.

We can tell if a device is on, off or standby mode based on the current between the wall socket and the device.

2nd Avatar - Energy Consumption Conserving:

Sensors	<ul style="list-style-type: none">• This agent is equipped with 6 presence sensors (one for every room), they could be either motion detection sensors in the entry of a room or we just call them presence sensors (there many ways of doing this).
Total 7	<ul style="list-style-type: none">• A sensor that receives the current time of this world.
Actuators	<ul style="list-style-type: none">• 6 power on-off actuators to control the lights.
Total 11	<ul style="list-style-type: none">• 3 ON/OFF/Standby actuators for the blu-ray player (media centre), sound system and TV. The standby is achieved by an off followed by an on (when he comes back home). The requirement here is that when a device is powered from off it will go to Standby mode. (Most are.)• 2 ON/OFF actuators for the dishwasher and the washing machine. When the washing is done it should be turned off and not remain on until someone comes and turns it off. This is achieved using a timer.

C-2

RoboHome GOLEM Agents:

1st Agent - Electricity consumption monitoring agent:

Description Monitor the Electricity Consumption

Goals	<ul style="list-style-type: none">• display in Real Time the Electricity consumption for every Device in every room and the total.• Record the Electricity consumption for every device in every room, the total cost and the time that events happen.• Provide information regarding electricity consumption to other agents using agent to agent communication protocols within GOLEM. This information includes: a) identification of the device consuming electricity b) identification of the location of the device c) information about the time that the consumption started and the time it ended d) the power consumption specification of the device and the cost (some devices consume more than others and at a different cost). This is also related to the time of consumption (day and night cost rates).• Graphically Display the graph Cost over time.
Sensors	<ul style="list-style-type: none">• Electricity consumption monitor for lights in SIX rooms (kitchen, living room, bedroom, corridor, toilet and laundry).• Electricity consumption monitor for TV (in use and standby).• Electricity consumption monitor for Sound System (in use and standby).• Electricity consumption monitor for Blu ray-media player (in use and standby).• Electricity consumption monitor for washing machine.• Electricity consumption of dishwasher.

Sensors that will not be used for the Avatar relationship:

- Message Listener, a listener for messages from other
-

agents within the MAS.

Actuators

- Message Sender, sends messages to other agents within the GOLEM MAS using GOLEM speech_acts.

2nd Agent - Consumption Conserving agent:

Description Save on electricity consumption

Goals

- When leaving the room, if light is left on and the user does not return within 10 minutes, it will turn the lights off
- When the user leaves the house or when he sleeps all devices go from standby mode to OFF. When he returns (or wakes up) they will go back to StandBy mode. Determine sleep if in bedroom for > 1 hrs (this excludes devices in bedroom).
- If the user forgets the thermostat on when he leaves the house it will turn it off.
- Communicate with other GOLEM agents to gather information regarding which device is consuming electricity at a given time, the duration of the consumption and the location of the device within the smart home. This information will be used to update the agents view of the system (variables describing the smart home device statuses). This view if the system is used by policies 1, 2 and 3.
- Inform User of actions. UI showing presence at any time and potential Actions taken.
- Record actions and time of actions.

Sensors

- This agent is equipped with 6 presence agent sensors (one for every room).
- A sensor that receives the current time of the real world. This will be useful for the calculations of how long something has been on etc.

Sensors that will not be used for the Avatar relationship:

- Message Listener. It is a listener for messages from other
-

agents within the MAS.

Actuators that will be used in the UA:

Actuators

- 6 power on-off actuators to control the lights.
- 3 ON/OFF/Standby actuators for the blu-ray player (media centre), sound system and TV. The standby is achieved by an off followed by an on (when he comes back home).
- 2 ON/OFF actuators for the dishwasher and the washing machine. When the washing is done it should be turned off and not remain on until someone comes and turns it off. This is achieved using a timer.

Actuators that will not be used in the UA:

- Message Sender, sends messages to other agents within GOLEM.

C-3

Simulation Values and Costs:

$$EC = (tmsec / 3600) * (itemWattage / 1000)$$

EC is the electricity consumption in kwh (kilowatt per hour).

tmsec the amount time that the light or device has been running in seconds divided by 3600 so we can get the hours.

itemWattage is the watt specification of the device divided by 1000 to get the kw that the device consumes.

And:

$$Cost = EC * (CostKWpH * / 100)$$

Cost is the cost of the electricity that has been consumed so far by the particular device.

EC is the electricity Consumption

CostKWpH is the cost in p of 1kw per one hour divided by 100 so we can get the cost in GBP.

Figure C-0-1. Calculation of electricity consumption and related costs for a device.

C-4

A description of the activities that are recognized and the UAs' reactions to them:

- a) If no motion activity is sensed by the camera sensor, the agent will control the light actuators in a way that the house appears to be habited to an external observer. In other words, it will switch them on and off regularly.
- b) If we position a toy-person inside the house, the agent will recognize the movement activity as sensed from the camera sensor and it will react to it by using the light actuator to turn the light on and the speaker actuator to request a password.
- c) It will then use the keyboard sensor to receive the password.
- d) If the password is incorrect or if a timeout expired before entering it, the following actions will be performed: i) the UA will use the speaker actuator to sound an alarm while also playing a pre-recorded message to discourage the intruder from remaining in the premises, ii) it will use the camera sensor to acquire a photo of the intruder and send it to the user and the "police" via the email actuator.
- e) Alternatively, if the keyboard activity is recognized as a successful input of the correct password, the UA will use the speaker actuator to welcome the toy-person and it will use the light actuators to perform light control management based on the toy-persons' location within the miniature house.

We notice that to create useful rules, we typically need to create complex activity recognition while taking into account an order of precedence regarding which rule will apply its actions when two or more rules are satisfied. For example to determine whether the toy-person is an intruder we would need to have already recognized both the movement activity and the wrong password activity leading to the deterring actions of d). On the other hand, if the

movement activity was detected for the first time and no password had been requested, the action would have been different (password request action).

C-5

The probability function of the “Random Number Generator” component will reach to a decision while taking into account of the weight that has been attached to the particular event. The weight is an integer between 1 and 100. UB is the upper bound indicating the weight for an event:

Integer UB, $UB \in (1, 100)$

The random variable X has the probability distribution N(1,100):

$X \sim N(1,100)$

N is a discrete uniform distribution where all values from 1 to 100 are equally likely to be observed (every one of n values has equal probability 1/100). If X is less than the Upper Bound UB then the event will occur otherwise it will be discarded. The equation below describes the logic behind the probability function.

Equation 4:

$X < UB?$ Event Occurs: Event Discarded

This means that if we attach a larger weight (upper bound) to an event it will be more likely for it to happen. The weights are decided by the system developer that uses the RoboHome platform.

The implementation uses the JAVA “Random” class to produce “pseudo-random” uniformly distributed integer values. (For more information, see the Random class documentation [122]).

C-6

UB is the upper bound for the duration of the particular event:

Integer UB

LB is the lower bound for the particular event:

Integer LB, LB < UB

The random variable D has the probability distribution $N(LB, UB)$:

Equation 5:

$D \sim N(LB, UB)$

N is a discrete uniform distribution where all values from LB to UB are equally likely to be observed (every one of n values has equal probability $1 / (UB-LB)$).

The D variable is the returned duration for the particular event.

The implementation uses the JAVA "Random" class to produce "pseudo-random" uniformly distributed integer values. (For more information, see the Random class documentation [122]).

APPENDIX D

D-1

The following table shows the numbers of agents, avatars and interactors that are being served by eVATAR as they increase in time during the execution of the test that was illustrated in Figure 7-1.

Time in seconds	Connections & Threads	Agents	Avatars	Interactors
60	2	1	1	2
6120	62	2	2	62
12180	122	3	3	122
18240	182	4	4	182
24300	242	5	5	242
30360	302	6	6	302
36420	362	7	7	362
42480	422	8	8	422
48540	482	9	9	482
54600	542	10	10	542
60660	602	11	11	602
66720	662	12	12	662
72780	722	13	13	722
78840	782	14	14	782
84900	842	15	15	842

REFERENCES

- [1] A. Artikis, M. Sergot, and G. Paliouras, 'A logic programming approach to activity recognition', *Proceedings of the 2nd ACM international workshop on Events in multimedia - EiMM '10*, 2010.
- [2] A. Artikis, A. Skarlatidis, and G. Paliouras, 'BEHAVIOUR RECOGNITION FROM VIDEO CONTENT: A LOGIC PROGRAMMING APPROACH', *International Journal on Artificial Intelligence Tools*, vol. 19, no. 02, pp. 193–209, Apr. 2010.
- [3] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni, 'Computational Logic Foundations of KGP Agents', *Journal of Artificial Intelligence Research*, p. 285, 2008.
- [4] A. Ricci, M. Piunti, and M. Viroli, 'Environment programming in multi-agent systems: an artifact-based perspective', *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 2, pp. 158–192, Sep. 2011.
- [5] A. Ricci, M. Viroli, and A. Omicini, 'CArtA gO: A Framework for Prototyping Artifact-Based Environments in MAS', *Environments for Multi-Agent Systems III*, pp. 67–86, Jan. 2007.
- [6] A. Saffiotti and M. Broxvall, 'PEIS ecologies', *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence innovative context-aware services: usages and technologies - sOc-EUSAI '05*, 2005.
- [7] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. S. Seo, and Y. J. Cho, 'The PEIS-Ecology project: Vision and results', *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sep. 2008.
- [8] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava, 'SensorWare: Programming sensor networks beyond code update and querying', *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 386–412, 2007.
- [9] A. Kansal, A. A. Somasundara, D. D. Jea, M. B. Srivastava, and D. Estrin, 'Intelligent fluid infrastructure for embedded networks', *Proceedings of the 2nd*

international conference on Mobile systems, applications, and services - MobiSYS '04, 2004.

[10] A. Rezgui and M. Eltoweissy, 'Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead', *Computer Communications*, vol. 30, no. 13, pp. 2627-2648, 2007.

[11] A. H. Ltd, 'The Architecture For The Digital World.' [Online]. Available: <https://www.arm.com/>. [Accessed: 13-Sep-2015].

[12] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, June-July 2003, pp. 317-323.

[13] B. Smith, "Reflection and semantics in a procedural language," *Massachusetts Institute of Technology, Cambridge, MA MIT-LCS-TR-272*, 1982.

[14] C. Cote, Y. Brosseau, D. L. Clement Raieovsky, and F. Michau, 'Robotic Software Integration Using MARIE', *International Journal of Advanced Robotic Systems*, p. 1, 2006.

[15] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, 'BASE - a micro-broker-based middleware for pervasive computing', *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003. (PerCom 2003)*., 2003.

[16] C. Bell, *Beginning Sensor Networks with Arduino and Raspberry Pi*. United States: APress, 2014.

[17] C.-L. Fok, G.-C. Roman, and C. Lu, 'Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications', *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[18] E. Aarts and R. Wichert, 'Ambient intelligence', *Technology Guide*, pp. 244-249, Jan. 2009.

- [19] E. Cerami, *Web services essentials: [distributed applications with XML-RPC, SOAP, UDDI & WSDL]*, 1st ed. United States: O'Reilly Media, Inc, USA, 2002.
- [20] E. M. Clarke and J. M. Wing, 'Formal methods: state of the art and future directions', *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, Dec. 1996.
- [21] E. Gamma, R. Helm, J. R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*, 1st ed. Reading, USA: Addison-Wesley Professional, 1995.
- [22] E. Marcus and H. Stern, *Blueprints for high availability designing resilient distributed systems*, 2nd ed. New York: John Wiley & Sons, 2003.
- [23] E. Cañete, J. Chen, M. Díaz, L. Llopis, and B. Rubio, 'A Service-Oriented Middleware for Wireless Sensor and Actor Networks', 2009 Sixth International Conference on Information Technology: New Generations, 2009.
- [24] E. Shakshuki, H. Malik, and M. K. Denko, 'Software agent-based directed diffusion in wireless sensor network', *Telecommunication Systems*, vol. 38, no. 3–4, pp. 161–174, 2008.
- [25] F. Bellifemine, A. Poggi, and G. Rimassa, 'JADE', *Proceedings of the fifth international conference on Autonomous agents - AGENTS '01*, 2001.
- [26] F. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*, 1st ed. United Kingdom: Wiley-Blackwell (an imprint of John Wiley & Sons Ltd), 2007.
- [27] F. Kon, F. Costa, G. Blair, and R. H. Campbell, 'The case for reflective middleware', *Communications of the ACM*, vol. 45, no. 6, Jun. 2002.
- [28] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, 'Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB', *Middleware 2000*, Jan. 2000.

- [29] F. Sadri, 'Ambient intelligence', *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–66, Oct. 2011.
- [30] F. Sadri and K. Stathis, 'Ambient Intelligence', in *Encyclopedia of Artificial Intelligence*, IGI Global, 2009, pp. 85–91.
- [31] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*, 1st ed. United Kingdom: Wiley-Blackwell (an imprint of John Wiley & Sons Ltd), 2007.
- [32] G. Acampora, D. J. Cook, P. Rashidi, and A. V. Vasilakos, 'A Survey on Ambient Intelligence in Healthcare', *Proceedings of the IEEE*, vol. 101, no. 12, pp. 2470–2494, Dec. 2013.
- [33] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, 'An Architecture for Next Generation Middleware', in *Middleware'98*, Springer Science + Business Media, 1998, pp. 191–206.
- [34] G. M. P. O'HARE, C. MULDOON, M. J. O'GRADY, R. W. COLLIER, O. MURDOCH, and D. CARR, 'SENSOR WEB INTERACTION', *International Journal on Artificial Intelligence Tools*, vol. 21, no. 02, 2012.
- [35] G. Aydin, 'Sensor Web Enablement. Open Geospatial Consortium (OGC)', 2004. [Online]. Available: <http://www.crisisgrid.org/html/ogc-swe.html>. [Accessed: 01-Sep-2015].
- [36] H. A. Kautz, 'Chapter 2 – A Formal Theory of Plan Recognition and its Implementation', *Reasoning About Plans*, pp. 69–125, 1991.
- [37] H. Sundmaeker, *Vision and challenges for realising the Internet of Things*. Luxembourg: Publications Office of the European Union, 2010.
- [38] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, 'A survey on sensor networks', *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, Aug. 2002.

- [39] *International Organization for Standardization/ International Electrotechnical Commission Standard ISO/IEC 7498-1: Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model (Second ed., corrected and reprinted 1996-06-15). Reference number ISO/IEC 7498-1:1994(E)*
- [40] J. C. Augusto and C. D. Nugent, Eds., *Designing Smart Homes*. Springer Berlin Heidelberg, 2006.
- [41] J. C. Augusto, J. Liu, P. McCullagh, H. Wang, and J.-B. Yang, 'Management of Uncertainty and Spatio-Temporal Aspects for Monitoring and Diagnosis in a Smart Home', *International Journal of Computational Intelligence Systems*, vol. 1, no. 4, p. 361, 2008.
- [42] J. M. Bradshaw, Ed., *Software agents*, 2nd ed. Cambridge, MA: AAI Press [u.a.], 1997.
- [43] J. Favela, M. Rodriguez, A. Preciado, and V. M. Gonzalez, 'Integrating Context-Aware Public Displays Into a Mobile Hospital Information System', *IEEE Transactions on Information Technology in Biomedicine*, vol. 8, no. 3, pp. 279–286, Sep. 2004.
- [44] J. Forth, K. Stathis, and F. Toni, 'Decision Making with a KGP Agent System', *Journal of Decision Systems*, vol. 15, no. 2–3, pp. 241–266, Jan. 2006.
- [45] J. M. Spivey, 'The Z Notation: a reference manual.' [Online]. Available: <http://spivey.oriel.ox.ac.uk/mike/zrm/>. [Accessed: 30-Sep-2015].
- [46] J. K. Ousterhout, *Tcl and the Tk Toolkit*, 8th ed. Reading, MA: Addison-Wesley, 1997.
- [47] J. L. Encarnação and T. Kirste, 'Ambient Intelligence: Towards Smart Appliance Ensembles', *Lecture Notes in Computer Science*, pp. 261–270, 2005.
- [48] J.M. Spivey, *The fuzz manual*, Computing Science Consultancy 34.
- [49] K. L. Calvert and M. J. Donahoo, *TCP/IP Sockets in Java: Practical Guide for Programmers*, 2nd ed. Amsterdam: Morgan Kaufmann Publishers, 2008.

- [50] K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In J. Muller and P. Petta, editors, *Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation" (AT2A1-4)*, pages 523-528, Vienna, April 13-16 2004.
- [51] K. Stathis and F. Toni, 'Ambient Intelligence Using KGP Agents', *Lecture Notes in Computer Science*, pp. 351-362, Jan. 2004.
- [52] K. Stathis, C. Child, W. Lu, and G. K. Lekeas. *Agents and Environments*. Technical report, SOCS Consortium, 2002. IST32530/CITY/005/DN/I/a1.
- [53] K. Römer, O. Kasten, and F. Mattern, 'Middleware challenges for wireless sensor networks', *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 4, pp. 59-61, Oct. 2002.
- [54] L. Capra, W. Emmerich, and C. Mascolo, 'CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications', *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929-944, Oct. 2003.
- [55] L. Richardson, S. Ruby, and D. H. Hansson, *RESTful Web Services: Web Services for the Real World*, 1st ed. United States: O'Reilly Media, Inc, USA, 2007.
- [56] L. Gurgen, C. Roncancio, C. Labbé, A. Bottaro, and V. Olive, 'SStreaMWare', *Proceedings of the 5th international conference on Pervasive services - ICPS '08*, 2008.
- [57] M. P. Papazoglou, *Web Services: Principles and Technology*, 1st ed. Harlow, England: Pearson/Prentice Hall, 2007.
- [58] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, 'Service-Oriented Computing: State of the Art and Research Challenges', *Computer*, vol. 40, no. 11, pp. 38-45, Nov. 2007.
- [59] M. D. Rodríguez and J. Favela, 'Assessing the SALSA architecture for developing agent-based ambient computing applications', *Science of Computer Programming*, vol. 77, no. 1, pp. 46-65, Jan. 2012.

- [60] M. Weiser, *The computer for the 21st century*, *Scientific American* 265. 3, 1991, pp. 94-104.
- [61] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *Knowledge Engineering Review*, vol. 10, no. 2, pp. 115-152, 1995.
- [62] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits, 'OASiS: A Programming Framework for Service-Oriented Sensor Networks', 2007 2nd International Conference on Communication Systems Software and Middleware, 2007.
- [63] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, 'A middleware infrastructure for active spaces', *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74-83, 2000.
- [64] M. Doddavenkatappa, M. C. Chan, and A. L. Ananda, 'Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed', in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer Science + Business Media, 2012, pp. 302-316.
- [65] M.D. Rodriguez et al., "Agent-Based Ambient Intelligence for Healthcare," *AI Comm.*, vol. 18, no. 3, 2005, pp. 201-216.
- [66] M.-M. Wang, J.-N. Cao, J. Li, and S. K. Dasi, 'Middleware for Wireless Sensor Networks: A Survey', *Journal of Computer Science and Technology*, vol. 23, no. 3, pp. 305-326, 2008.
- [67] N. Dipsis and K. Stathis, 'Internalizing Unknown Objects by Means of Perception and Communication in Multi-Agent Systems', in *Intelligent Environments*, 2009, pp. 499-509.
- [68] N. Dipsis and K. Stathis, 'EVATAR - A Prototyping Middleware Embodying Virtual Agents to Autonomous Robots', *Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010)*, pp. 167-175, Jan. 2010.

- [69] N. Dipsis and K. Stathis, 'Ubiquitous Agents for Ambient Ecologies', *Pervasive and Mobile Computing*, vol. 8, no. 4, pp. 562–574, Aug. 2012.
- [70] O. Vermesan and P. Friess, Eds., *Internet of Things - Global Technological and Societal Trends Smart Environments and Spaces to Green ICT*. Denmark: River Publishers, 2011.
- [71] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, 'The many faces of publish/subscribe', *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, Jun. 2003.
- [72] P. Grace, G. S. Blair, and S. Samuel, 'ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability', *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pp. 1170–1187, Jan. 2003.
- [73] P. Grace, G. S. Blair, and S. Samuel, 'A reflective framework for discovery and interaction in heterogeneous mobile environments', *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 1, p. 2, Jan. 2005.
- [74] P. Saint-Andre, 'Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence', 01-Jun-2004. [Online]. Available: <http://xmpp.org/rfcs/rfc3921.html>. [Accessed: 30-Sep-2015].
- [75] P. Vlacheas, R. Giaffreda, V. Stavroulaki, D. Kelaidonis, V. Foteinos, G. Poullos, P. Demestichas, A. Somov, A. Biswas, and K. Moessner, 'Enabling smart cities through a cognitive management framework for the internet of things', *IEEE Communications Magazine*, vol. 51, no. 6, pp. 102–111, Jun. 2013.
- [76] P. E. de Freitas, T. Heimfarth, C. E. Pereira, M. A. Ferreira, R. F. Wagner, and T. Larsson, 'Multi-Agent Support in a Middleware for Mission-Driven Heterogeneous Sensor Networks', *The Computer Journal*, vol. 54, no. 3, pp. 406–420, 2009.
- [77] P. Levis, N. Lee, M. Welsh, and D. Culler, 'TOSSIM', *Proceedings of the first international conference on Embedded networked sensor systems - SenSys '03*, 2003.
- [78] R. Giaffreda, 'iCore: A Cognitive Management Framework for the Internet of Things', *Lecture Notes in Computer Science*, pp. 350–352, Jan. 2013.

- [79] R. Iyer and L. Kleinrock, 'QoS control for sensor networks', *IEEE International Conference on Communications*, 2003. ICC '03., 2003.
- [80] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer, 'On the need for system-level support for ad hoc and sensor networks', *ACM SIGOPS Operating Systems Review*, vol. 36, no. 2, pp. 1–5, Apr. 2002.
- [81] S. Bahadori, A. Cesta, L. Iocchi, G. R. Leone, D. Nardi, F. Pecora, R. Rasconi, L. Scozzafava, S. Bahadori, A. Cesta, L. Iocchi, D. Nardi, F. Pecora, R. Rasconi, and L. Scozzafava, 'Towards Ambient Intelligence For The Domestic Care Of The Elderly', *Ambient Intelligence*, pp. 15–38, Jan. 2005.
- [82] S. Bromuri and K. Stathis, 'Situating Cognitive Agents in GOLEM', *Engineering Environment-Mediated Multi-Agent Systems*, pp. 115–134, Jan. 2008.
- [83] S. Bromuri, V. Urovi, and K. Stathis, 'Game-based e-retailing in GOLEM agent environments', *Pervasive and Mobile Computing*, vol. 5, no. 5, pp. 623–638, Oct. 2009.
- [84] S. Bromuri and K. Stathis, 'Distributed agent environments in the Ambient Event Calculus', *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems - DEBS '09*, 2009.
- [85] S. Franklin and A. Graesser, 'Is It an agent, or just a program?: A taxonomy for autonomous agents', in *Intelligent Agents III Agent Theories, Architectures, and Languages*, Springer Science + Business Media, 1997, pp. 21–35.
- [86] S. Vinoski, 'CORBA: integrating diverse applications within distributed heterogeneous environments', *IEEE Communications Magazine*, vol. 35, no. 2, pp. 46–55, 1997.
- [87] S. González-Valenzuela, S. Vuong, and V. C. M. Leung, 'A mobile code platform for distributed task control in wireless sensor networks', *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access - MobiDE '06*, 2006.

- [88] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, 'The design of an acquisitional query processor for sensor networks', *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*, 2003.
- [89] T. Erl, *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*, 1st ed. United States: Prentice Hall PTR, 2007.
- [90] T. G. Kim, H. Praehofer, and B. P. Zeigler, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd ed. San Diego, CA: Elsevier Science, 2000.
- [91] T. Liu and M. Martonosi, 'Impala', *ACM SIGPLAN Notices*, vol. 38, no. 10, 2003.
- [92] T.-H. Kim, S.-H. Choi, and J.-H. Kim, 'Incorporation of a Software Robot and a Mobile Robot Using a Middle Layer', *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 6, pp. 1342–1348, Nov. 2007.
- [93] W. J. J. Dixon and F. J. Massey, *Introduction to statistical analysis*, 4th ed. New York: McGraw-Hill Inc., US, 1983.
- [94] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo, 'Middleware to support sensor network applications', *IEEE Network*, vol. 18, no. 1, pp. 6–14, Jan. 2004.
- [95] X. Chu, T. Kobialka, B. Durnota, and R. Buyya, 'Open Sensor Web Architecture: Core Services', *2006 Fourth International Conference on Intelligent Sensing and Information Processing*, 2006.
- [96] Y. Shoham, J. M. Bradshaw, "An overview of agent-oriented programming", *Software Agents*, 1997 :AAAI Press/MIT Press
- [97] Y. Bai, H. Ji, Q. Han, J. Huang, and D. Qian, 'MidCASE: A Service Oriented Middleware Enabling Context Awareness for Smart Environment', *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*, 2007.

- [98] Y. Yu, B. Krishnamachari, and V. K. Prasonna, 'Issues in designing middleware for wireless sensor networks', *IEEE Network*, vol. 18, no. 1, pp. 15–21, Jan. 2004
- [99] [Online]. Available: [https://en.wikipedia.org/wiki/Avatar_\(computing\)](https://en.wikipedia.org/wiki/Avatar_(computing)). [Accessed: 31-Aug-2015].
- [100] 'Anatomy of Linux dynamic libraries', 20-Aug-2008. [Online]. Available: <http://www.ibm.com/developerworks/library/l-dynamic-libraries/>. [Accessed: 29-Sep-2015].
- [101] 'Apache MINA – Apache MINA.' [Online]. Available: <https://mina.apache.org/>. [Accessed: 27-Sep-2015].
- [102] 'Arduino.' [Online]. Available: <https://www.arduino.cc/>. [Accessed: 10-Aug-2015].
- [103] 'Class Hashtable<K,V>', 26-Sep-2014. [Online]. Available: <https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html>. [Accessed: 28-Sep-2015].
- [104] 'CORBA 3.3.' [Online]. Available: <http://www.omg.org/spec/CORBA/3.3/>. [Accessed: 28-Sep-2015].
- [105] 'European Commission: CORDIS: Projects & Results Service: Amigo Ambient Intelligence for the networked home environment', 06-Oct-2014. [Online]. Available: http://cordis.europa.eu/project/rcn/71920_en.html. [Accessed: 27-Sep-2015].
- [106] 'eVATAR – Miniature Robohome', 2015. [Online]. Available: <http://youtu.be/6bmKrL1Nf9Y>. [Accessed: 28-Sep-2015].
- [107] 'Extensible Markup Language (XML)', 2015. [Online]. Available: <http://www.w3.org/XML/>. [Accessed: 28-Sep-2015].
- [108] 'iCore homepage.' [Online]. Available: <http://www.iot-icore.eu/>. [Accessed: 28-Sep-2015].

[109] 'Inheritance (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance).' [Online]. Available:

<http://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>. [Accessed: 28-Sep-2015].

[110] 'Interface Queue<E>', 26-Sep-2014. [Online]. Available:

<https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>. [Accessed: 28-Sep-2015].

[111] 'Introduction – An Introduction to libuv.' [Online]. Available:

<http://nikhilm.github.io/uvbook/introduction.html>. [Accessed: 29-Sep-2015].

[112] 'Java Remote Method Invocation: - Contents.' [Online]. Available:

<http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/spec/rmiTOC.html>. [Accessed: 28-Sep-2015].

[113] 'Lesson: Packaging Programs in JAR Files (The Java™ Tutorials > Deployment).' [Online]. Available:

<http://docs.oracle.com/javase/tutorial/deployment/jar/>. [Accessed: 28-Sep-2015].

[114] 'Microsoft Windows DLL (Dynamic-link libraries)'. [Online]. Available:

<https://msdn.microsoft.com/en-us/library/twindows/desktop/ms682589>. [Accessed: 29-Sep-2015].

[115] 'OASIS SOA Reference Model TC', 03-May-2005. [Online]. Available:

https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm. [Accessed: 30-Sep-2015].

[116] 'OASIS Web Services Dynamic Discovery (WS-Discovery) Version 1.1', 01-Jul-

2009. [Online]. Available: <http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html>. [Accessed: 30-Sep-2015].

[117] 'OGC® PUCK Protocol Standard | OGC.' [Online]. Available:

<http://www.opengeospatial.org/standards/puck>. [Accessed: 18-Sep-2015].

- [118] 'OSGi Alliance | Main / OSGi Alliance.' [Online]. Available: <http://www.osgi.org/Main/HomePage>. [Accessed: 10-Aug-2015].
- [119] 'Overriding and Hiding Methods (The Java™ Tutorials > Learning the Java Language > Interfaces and Inheritance).' [Online]. Available: <http://docs.oracle.com/javase/tutorial/java/landl/override.html>. [Accessed: 28-Sep-2015].
- [120] 'OWL Web Ontology Language Overview.' [Online]. Available: <http://www.w3.org/TR/owl-features/>. [Accessed: 20-Sep-2015].
- [121] 'PPSP, open source statistical tool used to calculate standard deviations and plot distributions.' [Online]. Available: <http://www.gnu.org/software/pspp/>. [Accessed: 29-Sep-2015].
- [122] 'Random (Java Platform SE 6)', 30-Nov-2011. [Online]. Available: <http://docs.oracle.com/javase/6/docs/api/java/util/Random.html>. [Accessed: 28-Sep-2015].
- [123] 'Sensor Model Language (SensorML) | OGC.' [Online]. Available: <http://www.opengeospatial.org/standards/sensorml>. [Accessed: 19-Sep-2015].
- [124] 'SensorML.' [Online]. Available: <http://www.sensorml.com/>. [Accessed: 19-Sep-2015].
- [125] 'SOAP Specifications.' [Online]. Available: <http://www.w3.org/TR/soap/>. [Accessed: 30-Sep-2015].
- [126] 'Teach, Learn, and Make with Raspberry Pi.' [Online]. Available: <https://www.raspberrypi.org/>. [Accessed: 13-Sep-2015].
- [127] 'Telldus Technologies.' [Online]. Available: <http://www.telldus.se/>. [Accessed: 30-Sep-2015].

- [128] 'The C10K problem', 01-Jan-1999. [Online]. Available: <http://www.kegel.com/c10k.html>. [Accessed: 29-Sep-2015].
- [129] 'The ADAPTIVE Communication Environment (ACE).' [Online]. Available: <http://www.cs.wustl.edu/~schmidt/ACE.html>. [Accessed: 30-Sep-2015].
- [130] 'TinyOS.' [Online]. Available: <http://www.tinyos.net>. [Accessed: 10-Aug-2015].
- [131] 'UPnP-Universal Plug and Play Forum.' [Online]. Available: <http://www.upnp.org/>. [Accessed: 30-Sep-2015].
- [132] 'Web Service Definition Language (WSDL)', 01-Jan-2001. [Online]. Available: <http://www.w3.org/TR/wsdl>. [Accessed: 28-Sep-2015].
- [133] 'ZigBee® Wireless Standard - Technology - Digi International.' [Online]. Available: <http://www.digi.com/technology/rf-articles/wireless-zigbee.jsp>. [Accessed: 30-Sep-2015].
- [134] 'Z-Wave Alliance.' [Online]. Available: <http://z-wavealliance.org>. [Accessed: 30-Sep-2015].