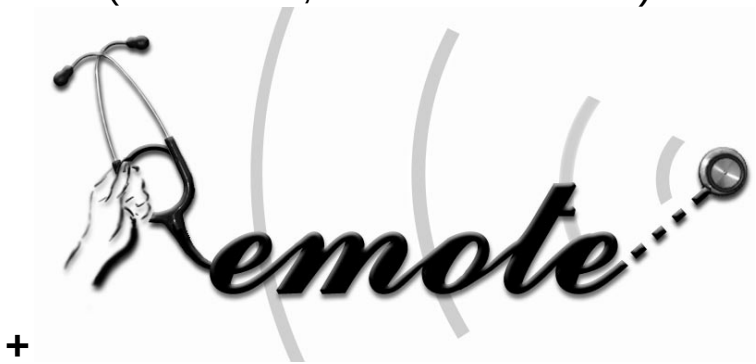
AMBIENT ASSISTED LIVING (AAL)
JOINT PROGRAMME

PROJECT

Remote health and social care for independent living of isolated elderly with chronic conditions

(REMOTE, AAL-2008-1-147)



DELIVERABLE

D6.1 REMOTE Services Methodology

WORK PACKAGE NO.	WP6	**WORK PACKAGE TITLE**	Integrated Services
TASK NO.	T6.1	**TASK TITLE**	Approaches and methodologies to building service-oriented
STATUS	D	**VERSION NO.**	v3
DOCUMENT ID	REMOTE_D6.1._REMOTE SERVICES METHODOLOGY		
FILE ID	REMOTE_D6.1._REMOTE SERVICES METHODOLOGY_v1.doc		
PROJECT START & DURATION	June 1st, 2009 36 Months		
AUTHORS[1]	Laura Pastor, María de las Mercedes Fernández-Rodriguez, María Fernanda Cabrera (UPM), Olga Gkaitatzi (CERTH/HIT), Verónica Crespo (ABAMA), Alejandro Aracil Ramón (TSB).		

[1] Per partner, if more than one partner, provide together

EXECUTIVE SUMMARY

This document presents the REMOTE Deliverable D.6.1 REMOTE Services Methodology.

It analyses several existing methodologies, specifying their elements and the design principles that need to be considered for the implementation of the REMOTE architecture.

As a result, SOA was selected as the most appropriate methodology and SoA framework to be used for the design and implementation of the REMOTE architecture.

Based on the Service Oriented Architecture (SOA) and the usage of a multi-agent architecture, several agents architecture have been analyzed.

Finally, different possible lifecycles have been considered, and the final usage of the IBM proposal determines the handling of the SOMA Methodology.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

REVISION HISTORY

Revision no.	Date of Issue	Author(s)	Brief Description of Change
1	11-02-2010	Veronica Crespo	Peer review of the document
2	18-06-2010	Ms. Olga Gkaitatzi	Peer review of the document

LIST OF ABBREVIATIONS AND DEFINITIONS

ABREV.	Abbreviation
ACL	Agent Communication Language
ALP	Advanced Logistics Program
ATP	Aglet Transfer Protocol
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DAS	Device Access Specification
DNS	Domain Name System
ERP	Enterprise Resource Planning
EU	European Union
FIPA	Foundation for Intelligent Physical Agents
IMTP	Internal Message Protocol
IP	Internet Protocol
IIOP	Internet Inter-ORB Protocol
HTTP	Hyper-Text Transfer Protocol
JADE	Java Agent Development Environment
J-ATCI	Java Agent Transfer and Communication Interface
JDK	Java Development Kit
JICP	JADE Inter-Container Protocol
JME	Java Micro Edition
JSE	Java Standard Edition
JVM	Java Virtual Machine
LAN	Local Area Network
LEAP	Lightweight Extensible Agent Platform
MASIF	Mobile Agent System Interoperability Facility
MIDP	Mobile Information Device Profile
MTP	Message Transport Protocol
MTS	Message Transport Service
OMG	Object Management Group
OS	Operating System
OSGI	Open Services Gateway Initiative
PC	Personal Computer
PDA	Personal Digital Assistant
QA	Quality Assurance
R&D	Research and Development
RFP	Request for Proposal
RMA	Remote Monitoring Agent
RMI	Java Remote Method Invocation
RPC	Remote Procedure Call
SDLC	Software development Lifecycle
SOAP	Simple Object Access Protocol
SOMA	Service Oriented Modelling and Architecture
TCP	Transmission Control Protocol
TRL	Tokyo Research Laboratory

UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UML	Unified Modelling Language
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
W3C	World Wide Consortium
WSDL	Web Services Description Language
XML	Extensible Markup Language

1 ARCHITECTURAL REFERENCE PARAMETERS IN INTELLIGENT AMBIENTS

This section proposes several types of architectures inside the software engineering that can be combined in order to get a final complete architecture for the Intelligent Ambient proposed, complying with all the REMOTE Project requirements. It is intended to provide a short explanation of the parameters defining the architecture. The following sections will specify the technical details of each one of them.

1.1 OPEN ARCHITECTURES

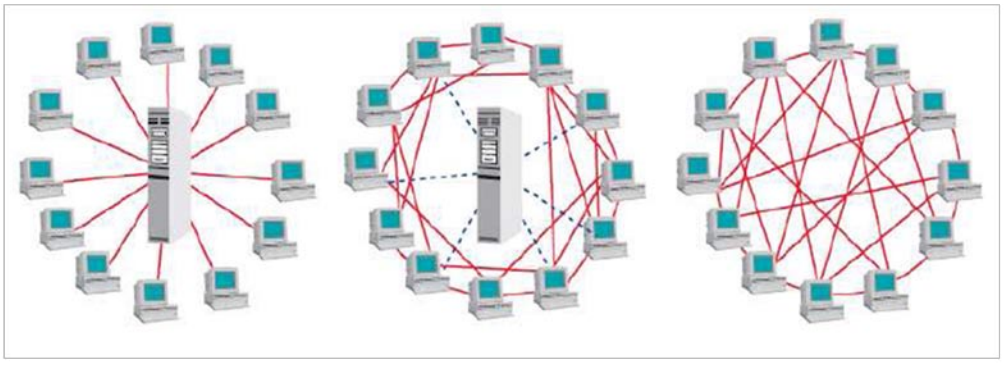
Open architectures are those that adopt a scalable and extensive solution. Open architectures can be extended after the system implementation, generally by adding additional circuits, for example, connecting the main system with a chip with a new microprocessor. System specifications are made public, which allows other organizations to manufacture the expansion products. In case of REMOTE project, due to the need of connecting a large variety of devices with different natures, it should be considered the possibility of being open to any new one that might arise. Even in the case of new applicable technologies not considered at the beginning of the project but that may appear during it.

1.2 DISTRIBUTED ARCHITECTURES

Distributed systems are generally the best way to implement network systems, even though they imply several disadvantages, all of them derived from the network security control.

In our case, each device will capture and process signals, acting as client and the middleware1 that is in charge of redirecting them, will act as a server. A distributed architecture will provide the system with a high scalability. If the number of devices increases, the system should also allow be extended in a transparent way for the units that already produced.

There are three possible topologies concerning a distributed architecture:



a) Client/Server b) Hybrid Peer to Peer c)Peer to Peer

Figure 1- Topologies of distributed Architectures

The concept of middleware is related to as the intermediate software layer that provides support to the specific services and applications. Middleware should not be confused with a specific implementation of the architecture, given that this is a component of it.

1.3 SERVICE ORIENTED ARCHITECTURES (SOA)

Irrespective of the technology chosen, in order to implement our architecture, simultaneously we can provide it with the features of another complementary architecture such as the Service Oriented Architecture. This architectural conjunction will report a set of very desirable features when covering the Project needs.

There is an existing set of guidelines defined by Thomas Earl in the book "SOA" broadly supported by the industry that makes certain architectures Service Oriented:

- Formal Contract establishment
- Abstraction of the service
- Low coupling between services
- Reusable Services
- Autonomous Services
- Services without status
- Services should be able to be discovered
- Possibility of service composition

By means of this architecture, it is possible to find several devices that offer services to the end users (or other services or applications).

Starting from a series of basic services that provide elemental functionalities, service composition allows aggregating them in a way so a higher level of services is achieved, therefore better adapted to the user needs. Consequently, service composition intends to complete the existing space between basic functionalities and requirements proposed by the user.

SOA development implies a series of practices in the architecture such as the weak coupling among services, strongly into layers, separation of responsibilities and integration. Therefore, it is not a product but a way to get to the architecture. SOA intends to achieve the granularity of the business service adapted to the needs of the business unit of the Organization, so there is an alignment of the business processes with the services. The main characteristic that differentiates the last ones from others is that they are not focused on an interface, but in a contract established between the schemas communicating with each other by exchanging messages.

If the platform is not designed based on standards and directed to the interoperability, it is complicated to quickly combine services together in order to comply with the requirements of the continuous business change.

1.3.1 Web Services

Web Services are simply an implementation of the SOA architecture that complies with its nature. The World Wide Web Consortium defines it as "A Software application identified by an URL, whose interfaces can be defined, described and discovered through XML documents. A Web service supports direct interactions with

other software agents using XML messages exchanged through Internet based protocols."

Web Services are currently experiencing an important evolution fostered by the new concepts of Web 2.0 and Web Semantic. Next figure illustrates the three generations that are assumed for the Web Services for the next years.

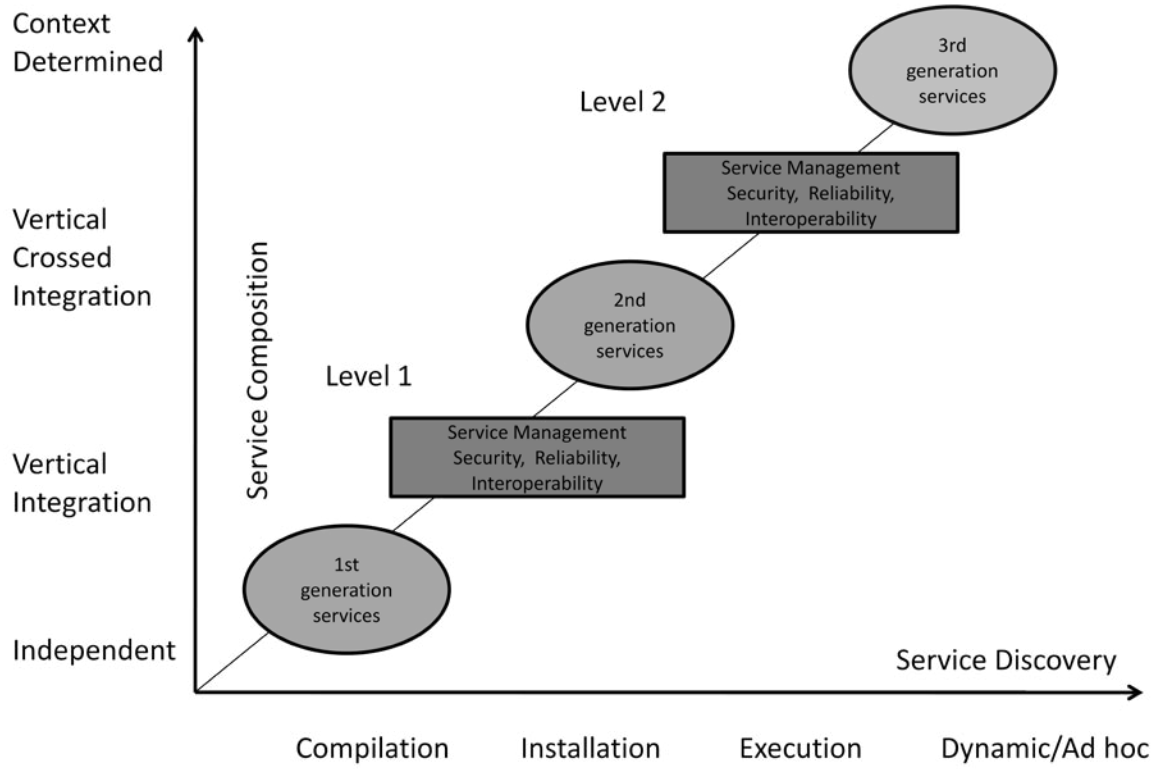


Figure 2- Web Services Evolution

1.3.2 Semantic Services Oriented Architectures

Web Services add a new functionality level to the current Web, being a first step to achieve the integration of distributed software components using Web standards. However, the current technology they are dealing with, such as the use of SOAP, WSDL and UDDI, operates in a syntactic level (some authors call it Syntactic Web), and even though interoperability among them is considered through common standards, they still require human interaction. Generally the major disadvantage of Web Services is the lack of dynamism in the service composition. It is very common to work with a central control point that establishes the services flux and activates the services following this flux.

Apparently this is not the best implementation approach in order to adapt to a dynamic decentralized environment such as the one that is assumed for REMOTE in particular and to Intelligent Ambients in general, where service composition through semantic services is presented as a promising approach.

In fact, being able to achieve this goal, describing the Web services in this way would allow the cooperation between several systems and organizations in a natural way, with a minimum external intervention. Besides, this paradigm would imply that the web services provided by a certain organization could be located and used by other

services and/or applications, from the same organization or from other ones, so we could compose much more complex systems.

Therefore, this architecture appears as a natural evolution of the architectures implemented through Web services, and therefore Service Oriented.

Web Semantic Services are a step forward to achieve a dynamic composition and discovery of services, where intelligent systems intent to build services from the user requirements, without a manual selection of the services. They are built over knowledge representation technologies, with ontologies describing the domain in a formal way, and Ambient Intelligence providing methods in order to make the service composition more autonomous.

Ontologies come from the field of Artificial Intelligence. They are common vocabularies for the people and the applications that work in a domain. According to the Ontologies Working Group of the W3C Consortium, ontology defines the term that is used to describe and represent a certain domain. The term "domain" is used to denote a specific interest area (hospital, for example) or a knowledge area (domotics, sensor network, medicine, accounting, product manufacturing, etc.). Each ontology represents a certain vision of the world with respect to a domain.

Every person has in mind ontologies that are used to represent and understand the surroundings. These ontologies are not explicit, in the sense that they are not detailed in a document neither organised in a hierarchic or mathematic way. It is not necessary to make this knowledge explicit, because it is part that everybody knows. However, when dealing with non common terminology or when these terms need to be processed by machines, it is needed to specify the ontologies in a document, providing them with a format intelligible for the machines.

Machines lack from the ontologies that humans have in order to understand the world and communicate among them; that is why explicit ontologies are needed. When two information systems (ERP systems, data bases, knowledge bases) intend to communicate together, semantic problems that make difficult or impossible the common understanding and the communication between them appear. There are two types of semantic problems: domain and name related. Domain conflicts appear when similar concepts with regards to their meaning, but not identical, are represented in different domains (for example Doctor in a hospital or at the university). Regarding the name conflicts there are two types: synonymous and homonymous. Synonymous occurs when the systems use different names in order to refer to the same concept (for example employer and employee). Homonymous appears when the systems are using the same name to represent two different things (for example driver in an insurance company could be the named insured and the person that performs the transportation). These understanding problems that can arise from a communication established between Web Services are solved if the services definition is done in a semantic way.

Besides, ontologies are very useful in order to facilitate automatic reasoning, that is, without human intervention. Starting from some inference rules, a reasoning engine can use the ontologies in order to infer conclusions about them.

2 SERVICE AND MIDDLE WARE ORIENTED ARCHITECTURES

This section presents the principles that need to be taken into account in order to design a Service Oriented Architecture.
Following, service registry and search processes are described, as well as the service composition feature, provided by SoA.
In order to complement the usage of the SoA, several agent architectures are described, and their pros and cons are analyzed.

2.1 SoA Design principles

The design principles of SOA according to an industry expert Thomas Erl defined in his book "SOA: Principles of design are: Standardized Service Contract, Service Loose Coupling, Service Reusability, Service Autonomy, Service Statelessness, Service Discoverability, Service Composability and Service Abstraction.

2.1.1 Standardized Service Contract

Standardized service contract principle states that all services contained in the same inventory of services must be in compliance with the same contract designs standards. The aim of this principle is to ensure the coherent expression of the general purpose and functionalities of the service by naming conventions and consistent expression.

Standardization makes services easier to understand and use. Consistency reduces understanding efforts and increases the interpretability about what the service can do and how these features can be accessed.

Thomas Earlf defines the following three types of Service Contract Standardization:

- **Standardization of Functional Service Expression:** This refers to establish conventions for describing the functionalities of the service.
- **Standardization of Service Data Representation:** This refers to establish conventions for describing the data types and defines schemas for each information set. Defining schemas provides a mechanism so that different services can use the same data types while the interoperability is improved.
- **Standardization of Service Policies:** This refers to the property vocabulary used to express business rules, which should be standardized to avoid confusion for consumers of the service. This principle is closely related to the rest.

An illustration of a bad and a good way to apply the above mentioned principles follows:

Bad	Good
<message name="GetItemRequest"> <part name="**RequestValue**" element="bus:ItemIdentificator"/> </message> <message name="GetItemResponse"> <part name=" **GetItemResponse** " element="bus:Item"/>	<message name="GetInvoiceRequest"> <part name="RequestValue" element="bus:InvoiceNumber"/> </message> <message name="GetInvoiceResponse"> <part name="ResponseValue"

</message> In this case there are no conventions on the names. If the name of the input is RequestValue it is expected that the name of the output is ResponseValue.	element="bus:Invoice"/> </message>

2.1.2 Service Loose Coupling

Service loose coupling principles states that all services should be independent of the rest of the components of the solution. This principle aims at establishing uncoupled services reducing the dependence between the service contract, its implementation, and its service consumers.

If a contract has been well designed, the changes of implementation should not affect it. In the same way, as far as the contract has not been modified, the implementation updates should not produce changes in the service consumption.

The main dependences that can be found among the elements of a SOA are: the relation between the business process and the logic, the relation between the logic and the implementation, the relation between the logic and the contract and the relation between the contract and the consumers.

One recommendation in order to reduce the contract dependence with the logic consists of making a first contract before starting logic solution to keep the contract complies with an implementation. In this sense, the logic fulfills contract. The problem appears when a contract is derived from a logic solution, in this case each time the logic changes, a new contract is built.

2.1.3 Service Abstraction

Services contracts only contain essential information about the services, and these are limited to comply with the public information in the contract. The Service Abstraction principle emphasizes the need to hide the details of a service as much as possible.

Service's consumers do not need to know which database or which programming language has been used. These details unnecessary distract consumers from focusing on what really matters, understanding the service's use.

Abstraction favours implementation's changes without major impacts. As the service can be invoked, it should be specified how to utilise its functionality it and its use limitations, such as the maximum number of concurrent users and the maximum usage time. Details such as the programming language used, the functions called by the implementation or the last error thrown are additional data that must be hidden.

Following, a good and a bad example of the information about a service that needs to be specified are provided.

	Bad	**Good**
Metadata	The service is developed with Java and connected	The service is invoked using SOAP.

	to MySQL.	
Functional	Includes internal functions like validating user, connecting to database, management pool connection. Includes functions not available to consumers like private addItem (item, user).	Includes public functions like addItem (item), deleteItem (item), searchItems.
Quality	The connection pool maintains open 4 connections.	The service is only available during 6 hours a day. The service only supports 40 users concurrently.

2.1.4 Service Reusability

This principle states that all services must be designed and built thinking about reusing them within the same application, within the domain of enterprise applications or even within the public domain for massive use. This principle raises numerous design considerations to guarantee that individual service capabilities are properly defined in relation to an independent service context, and also that the necessary reuse requirements are facilitated.

Even if a service is built around a business process, service design must be general, not just focused on a single consumer, designers should determine the potential consumers. In this sense, the service must be designed to be invoked by different consumers.

In order to avoid implementing a service already existing and properly performing, the service should be made publicly available.

Example: We should imagine a scenario of a programmer that has been hired to design and implement a service to manage a store that sells clothes. This programmer thinks that the service needs to be able to add, delete and search garments by category, so he needs to define a data type called “clothing”, covering all types of garments. This way of thinking may cover more or less the needs of any clothing store, however, all the stores manage inventories and need to be aware of what they currently have, and need to update this stock with the purchase and sale of products. If the initial service, instead of using a data type called “clothing” uses a generic type called “item,” the data type “clothing” would be a particular case of “item” and the service can be used for managing the inventory of potentials stores.

2.1.5 Service Autonomy

Service autonomy SOA design principle states that services must have a high level of control over their underlying runtime and execution environment. In this sense, every service must have its own runtime environment. In this way, services are totally independent and it can be ensured that they can be reused from the execution platform point of view.

Thomas Earlf defines the following categories of service Autonomy:

- **Service Contract Autonomy:** This refers to two services that can not share the same functionalities.
- **Shared Autonomy**: This occurs when the resources are shared with other services or other parts of the architecture.
- **Service Logic Autonomy**: This happens when the logic components of the service are dedicated, but other resources like databases are shared.
- **Pure Autonomy:** This occurs when all resources are dedicated and can be isolated.

2.1.6 Service Statelessness

Service statelessness principle states that services should avoid resource consumption by managing state information only when necessary. The management of state information affects the scalability and availability of a service. According to this principle services are designed to have state information only when needed.

A reduction of the managing load of the state information leads to a higher degree of service reusability, because in this way service would be more generic.

A service is in an active state when it is invoked or executed, otherwise it is in a passive state when it is not in use. It can be stated that a service is stateless when it is in an active state but it is not processing the state data. However, if the service is active and it is processing the state data, the service is stateful.

There are three different types of state data: session data, context data and business data. Session data are those stored during the communication with the client; context data refers to the data exchanged between services and business data are the ones associated with the persistence data.

Handling large amounts of data could be done in a progressive way, therefore only using the data needed, so scalability is not compromised. However, if the service has pure autonomy, it can use a database in order to store and retrieve temporal activity information and being kept stateless as long as possible.

2.1.7 Service Discoverability

Communicative Metadata facilities find and interpret correctly the services. The service discoverability principle states that every service must be able to be discovered in some way so it can be used, so the accidental creation of services providing the same functionality can be avoided.

Service registry is used to support discoverability and interpretability of the services. This mechanism sets formalities to facilitate the location and recovery of a service through Metadata. The discoverability refers to the ability of detection and interpretability refers to the ability to be understood.

This principle is strongly linked with the principle of Service Reusability, because only if a service is found and understood in a correct way, it can be avoided to implement again something already existing.

2.1.8 Service Composability

Service composability design principle states that all services are effective composition participants, unaffected by the size and complexity of this composition.

In this sense, all services must be built considering that they can be used to implement generic higher level services, based on lower-level services.

In order to guarantee that a service can be used in a composition, it is necessary to assure that the concurrence of users and the data types of contract are as generic as possible, offering diverse possibilities to access the same type of functionalities.

2.2 SERVICES REGISTRY AND SEARCH

One of the premises of the Service-Oriented Architecture is that services should be accessible from outside. In this sense, there should be a place where other applications can know which services are available and how to access them. This place is called **Services Repository**. All services available in the system should be registered in the services repository, allowing anyone to make a consultation and if needed, make the discovery of them.

In order to register the service in the services repository, a formal contract is needed. This contract specifies the details for the service search. A further evolution of this discovery method is based on a semantic modelling, providing much more useful and accurate information about the service to be discovered.

Service linking with a web service can be static or dynamic. It is static when the link with the service happens at design time. However, it is dynamic when the link with the service happens at runtime.

UDDI organizations defined a standard for services registries, which is an intermediary between consumers and service providers. The types of information stored in an UDDI structure are the following:

- **BusinessEntity:** includes the necessary information to describe the provider of the service.
- **BusinessService:** refers to non-technical information associated with a service.
- **BindingTemplate:** includes the technical information necessary to access the service such as the URL.
- **tModel:** refers to the technical model.

When registering a service, it is common to use the different types of UDDI registries, for example one internal to the organization, a registry that belongs to partners and a public node management in companies like IBM.

The search of services is facilitated by the use of UDDI registries, grouping services by the business entity or by categorizations.

2.3 SERVICES COMPOSITION

The composition of services is a feature acquired by Service Oriented Architectures once they are ready to be discovered. The goal of service composition is to create different layers of services in which each layer is based on the composition of the above services. Thus a dynamic and independent system in which a service can be replaced in a very versatile way with a new composition of its component services is obtained.

Depending on the proximity to business processes, there are different levels of services. It usually exists a first level called domain level, which encompasses all those services related to a specific domain of business; the second level is a business process level, where clearly defined business processes that can be pursued are located, their own business services that include operations common to all business processes and last infrastructure services where communication services, event services, discovery services, similarity search services, logging services or services in context can be found .

In a services composition, services have two roles depending on the service's position in the configuration. A service is a controller when it is the head of the composition. In this case, the capabilities of the controller will use capabilities of other services. On the other hand, if a service is used as an element of the composition, it is called composition member. The capabilities of a composition member are invoked.

When the composition is very simple it is called primitive composition. In this case few services are involved and the relationships between them are very simple. However, when the configuration of the composition is more convoluted and requirements are more compound, so we are talking about complex service compositions, a mature service inventory is required.

2.4 DEVICES SEARCH

Under the REMOTE project, devices need to make requests to services. In this sense, it is important to consider that technologies may be used so the devices can connect and invoke the services.

The first technology to be considered is UPnP, which defines an architecture that allows connections from smart appliances, wireless devices and computers in all possible combinations. This architecture provides a distribution network that promotes open access using technologies such as TCP / IP allowing data transfer devices.

The UPnP Device Architecture is designed thinking of using a zero-configuration and to automatically discover a variety of devices. In this sense, devices can join the network, obtaining an IP address, share their capacity and discover other devices that are present. The device is able to disconnect from the network seamlessly when desired.

UPnP uses Internet protocols such as TCP, IP, HTTP, UDP, and XML. It uses a declarative language expressed in XML to create the contracts, which are communicated through HTTP. The advantage of this architecture is that it does not use drivers, but common protocols. UPnP is also independent of the programming language and the operating system.

Another technology to be considered is the one developed by Sun Microsystems called Jini, which provides a mechanism to connect devices to a network without the need to be previously configured. Jini also has a discovery process of devices to detect when they are connected to the network allowing them to report on their capabilities. After executing the discovery process an IP address is provided to the device in the network.

In Jini, each device provides to the rest of devices information about the services, driver and interfaces that they use. This architecture is fully distributed and no device is a controller or a root network. There are no limitations to the constant connection and disconnection of devices. Jini architecture does not need a central computer to control the devices connected to the network.

The above mentioned technologies are contained in the internal structure of OSGI, which acts as the lowest layer of service that is able to detect devices that are within the scope of the project. This is what will enable devices to connect and invoke services of the REMOTE project.

OSGI aims at defining open specifications for designing software compatible platforms that can provide multiple services. Although OSGi defines its own architecture, it has been designed for compatibility with Jini or UPnP. The OSGi specification has been defined with a set of APIs for developing basic services, such as logging, HTTP server and the Device Access Specification (DAS), which allows devices to discover the services offered by them.

One of the fundamental elements of the OSGI architecture is the Service Platform, located on the local network and connected to the service provider through a gateway at the operator's network. This element is responsible for enabling

interaction between devices or networks of devices using different technologies to communicate.

Finally, the remarkable benefits of using OSGi are:

- The services can be removed or added without rebooting the system.
- It is possible to create services providing increased functionalities to the initial ones.
- OSGI has its own security layer which allows increasing the security of Java setting permissions to register or access services.
- Libraries are shared, which allows memory savings.
- It checks dependences and avoids execution conflicts.
- Several versions of the same service can coexist.
- It enables life cycle management of services. OSGI can also monitor and react to life cycle changes in the service.

2.5 AGENTS ARCHITECTURES

Agent's architectures are formed by a series of components defined by P. Maes as "Computational systems that are located in complex dynamic environments perceive and act in an autonomous way, developing a set of tasks and fulfilling objectives for the ones they were designed".

According to the artificial intelligence principles, an agent has the following properties: autonomy, sociability, reaction capacity, initiative, benevolence and rationale.

If a system following these principles has two or more intelligent agents, it becomes multi-agent.

Following, some examples of relevant multi-agent architectures are provided.

2.5.1 The JADE Platform

2.5.1.1 Introduction

The Java Development Framework (JADE) is a middleware for the development and run-time execution of peer-to-peer applications which are based on the agents' paradigm and which can seamless work and interoperate both in wired and wireless environment.

It is used as a higher-level library that enables easier and more effective application development by providing useful services for a variety of applications, such as information access, encodings, communication and resource control. JADE has been implemented fully in Java language and it can be seamless executed on every typical type of Java Virtual Machine.

Moreover, "JADE is used for the development of distributed multi-agent applications based on the peer-to-peer communication architecture. The intelligence, the initiative, the information, the resources and the control can be fully distributed on mobile terminals as well as on computers in the fixed network. The environment can evolve dynamically with peers that in JADE are called agents that appear and disappear in the system according to the needs and the requirements of the application environment".

2.5.1.2 JADE architecture

Figure 3 shows the main architectural components of a JADE platform. A JADE platform is composed of agent *containers* (agents' groups) that can be distributed over the network. Agents live in containers which are the Java processes that provide the JADE run-time and all the services needed for hosting and executing agents. There is one special container, called the *main container*, which represents the bootstrap point of a platform: it is the first container to be launched and all other containers must register with it. The diagram in Figure 3 schematizes the relationships between the main architectural elements of JADE.

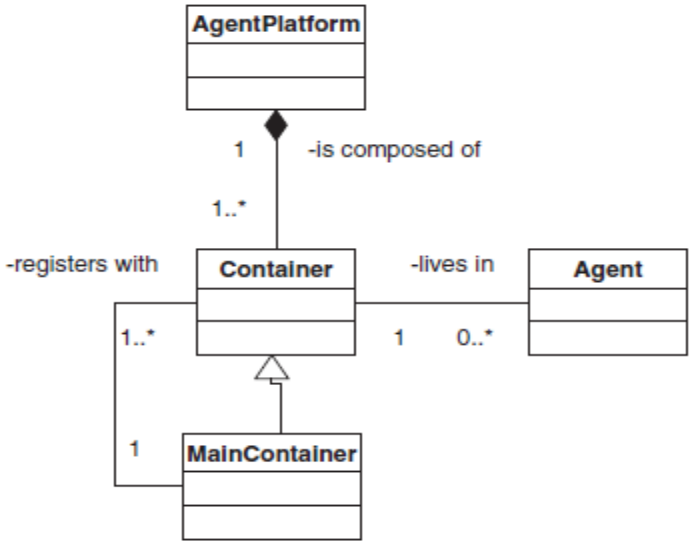


Figure 3 - Relationship between the main architectural elements

The programmer identifies containers by simply naming them; by default the main container is named 'Main Container' while the others are named 'Container-1', 'Container-2', 'Container-3' etc. The Context Broker Architecture (CoBrA) will be based upon JADE architecture. The CoBrA will use and take advantage of JADE facilities and moreover it may extend some of them.

2.5.1.3 Running JADE Agents on Devices

JADE agents can run on multiple and different type of devices. They can run on smart phone devices, on PDAs or PCs. JADE provides additional libraries (add-ons) in order to support it. The LEAP add-on replaces certain parts of the JADE kernel, forming a modified run-time environment that is identified as JADE-LEAP and which can be deployed on a wide range of devices.

JADE-LEAP can be shaped in three different ways corresponding to the three main types of Java environments:

- ***JSE***: to execute JADE-LEAP on PCs and servers in the fixed network running JDK1.4 or later.
- ***pJava***: to execute JADE-LEAP on hand-held devices supporting JME CDC.
- ***MIDP***: to execute JADE-LEAP on hand-held devices supporting MIDP such as Java-enabled CLDC smart phones.

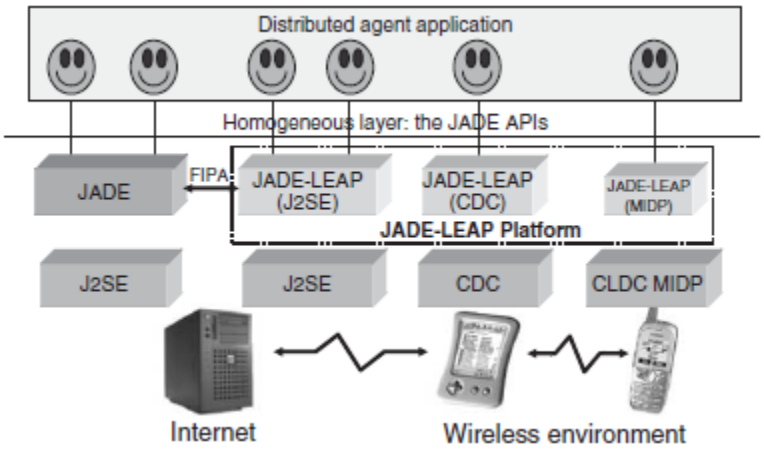


Figure 4 - The JADE-LEAP run-time environment

The developers can generally deploy JADE agents on JADE-LEAP and vice versa without changing the code. However, JADE containers and JADE-LEAP containers cannot be mixed within a single platform. JADE-LEAP platform and a JADE platform can, of course, communicate as specified by FIPA, e.g. by using the HTTP Message Transport Protocol.

2.5.1.4 Message Transport Service

The Message Transport Service is responsible for managing all message exchange between peers.

2.5.1.5 MTP

To promote interoperability between different platforms, JADE implements all the standard **Message Transport Protocols** (**MTPs**) defined by FIPA, where each MTP includes the definition of a transport protocol and a standard encoding of the message envelope. JADE always starts an HTTP-based MTP when initialising a main container; there is no MTP activated on normal containers. This creates a server socket on the main container host and listens for incoming connections over HTTP. Internally, the platform uses a transport protocol called IMTP (Internal Message Transport Protocol) which is described in the next section.

2.5.1.6 IMTP

The **JADE IMTP** (**Internal Message Transport Protocol**) is solely used for exchanging messages between agents living in different containers of the same platform. It is different from inter-platform MTPs, such as HTTP. As it is only used for internal platform communication, it does not need to be compatible with any FIPA standards. The JADE IMTP is not only used to transport messages, but also to transport internal directions essential to manage the distributed platform, as well as to monitor the status of remote containers. For example, it is used to transfer a command to order shut down a container.

JADE was designed to allow selection of the IMTP at platform launch time. Up to date, two main IMTP implementations are available. One is based on Java **RMI** and is the default option. Using RMI it is possible to invoke methods in a remote way so the communication between servers in distributed applications can be carried out based on Java. The second one is based on a proprietary protocol using **TCP** sockets that circumvents the absence of support for Java RMI in the JME environment; this is started by default when initiating the JADE LEAP platform.

2.5.1.7 THE LEAP IMTP

Though JADE and JADE-LEAP are almost identical from an external point of view, they are quite different internally. The normal JADE IMTP is based on *Java RMI* and this is not suitable for mobile devices. JADE-LEAP therefore uses an alternative IMTP based on a proprietary protocol called *JICP* (*Jade Inter Container Protocol*). The difference between the two protocols (RMI and JICP) is the main reason why a JADE-LEAP container cannot register with a JADE main container.

2.5.1.8 JADE execution modes

The following table summarizes how the two execution modes ('*stand-alone*' mode and *split* mode) are supported in different environments targeted by JADE-LEAP.

	J2SE	CDC	MIDP
Stand-alone	Suggested	Supported	Not Supported
Split	Supported	Suggested	Mandatory

Table 1 - JADE-LEAP Execution modes

The split mode has been successfully applied to the ASK-IT EU project, while the standalone mode is supported by Windows Mobile devices which have a CDC configuration. These devices run an independent JVM such as J9, Jbed etc. and are also able to support the split execution mode.

When launching the JADE run-time using the *split* execution mode, the user is not creating a normal container (as in stand-alone mode), but a very thin layer called the *front-end*. The front-end provides agents with exactly the same features of a container. However, it implements only a small subset of them directly, while delegating the others to a remote process called the *back-end*. The front-end that runs in the device is more lightweight than a complete container.

Figure 5 - Stand-alone execution mode

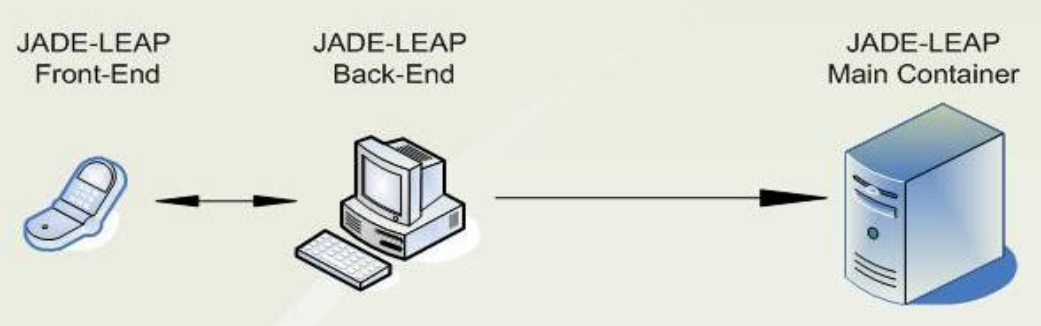


Figure 6 - Split execution mode

If the connection between the front-end and the back-end goes down (e.g. In case the cell-phone host enters a dead spot), messages to and from agents living in the front-end are buffered in both the back-end and the front-end. As soon as the front-end re-establishes a connection, buffered messages are delivered to their receivers.

The IP address of the mobile device is never seen by other containers in the platform, since they always interact with the back-end. As a result, the IP may change without any impact to the application.

2.5.1.9 Admin and Debugging JADE Tools

There are various tools provided with the JADE framework for administrative purposes. The most important of them are the following:

- **The platform management console** or the JADE RMA (Remote Monitoring Agent) is a tool that implements a graphical platform management console.
- **The Dummy Agent** is a very simple tool sending ACL messages to test the behaviour of another agent.
- **The Sniffer Agent** is a tool used to sniff, monitor and debug agents' conversations.
- **The Introspector Agent** is a tool used to debug a single agent.
- **The Log Manager Agent** allows the logging levels of each JADE platform component to be changed at run-time.

The Event notification Service and the JADE Tool Model is a platform-level service interpreting and routing generated events (e.g. agent born, message sent) to all agents subscribed to receive notifications

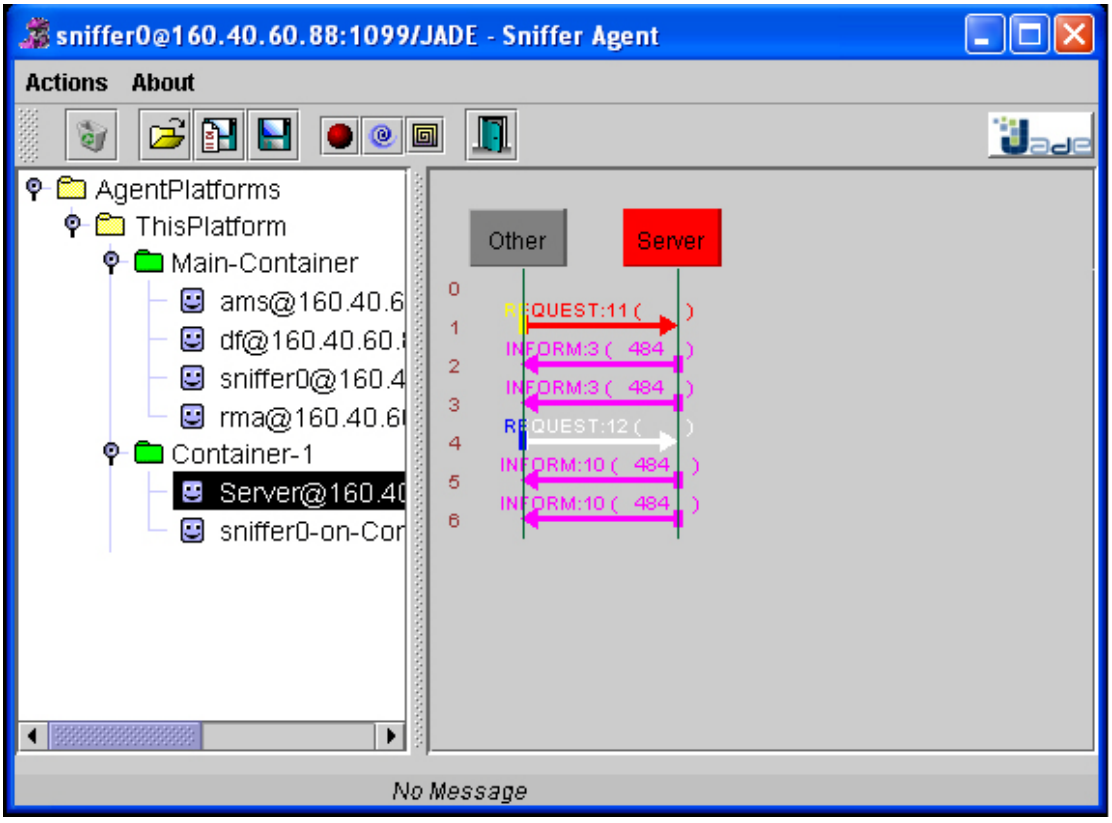


Figure 7- JADE Sniffer Agent

The JADE Sniffer Agent can be used to view the message exchange between agents, as it is depicted in above figure.

2.5.1.10 JADE Conclusions

"JADE provides the basic services necessary to **distributed peer-to peer** applications in the fixed and mobile environment. JADE allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm. From the application point of view, each agent is identified by a unique name and provides a set of services. It can register and modify its services and/or search for agents providing given services, it can control its life cycle and, in particular, communicate with all other peers".

Agents communicate by exchanging **asynchronous messages**, a communication model almost universally accepted for distributed and loosely-coupled communications, i.e. between heterogeneous entities that do not know anything about each other.

JADE can be applied on **limited constraint devices** by providing specific add-ons. It supports scalability by providing the opportunity of executing multiple parallel tasks within the same Java thread.

It supports **mobility** of code and of execution state. An agent can stop running on a host, migrate on a different remote host and restart its execution from the point it was interrupted.

Furthermore, it supports **yellow-paging** by advertising services that can be distributed across multiple hosts.

Although JADE has been successfully used on peer-to-peer (client/server) distributed system architecture in many commercial products and EU projects, it has not been used for R&D yet.

2.5.2 AGLETS (Java library for mobile agents)

2.5.2.1 Introduction

Aglets have been developed at the IBM Tokyo Research Laboratory (TRL) from Mitsuro Oshima and Danny Lange. They were responsible for the most of the 1.x releases. However version 2.x is totally open source and can be found at Sourceforge.net.

"Aglets have been immediately involved in the realization of *TabiCan*, a kind of virtual agent-populated travel agency. Unfortunately, after a good start, IBM decided to give Aglets to the open source community, and this is when SourceForge appears. In the beginning, the SourceForge releases have been only bug-fix ones, but then something changed and the library version evolved to the 2.x series. The 2.x thread has new improvements in the security management, and it is more compatible with the Java 2 security mechanism than the 1.x releases. Furthermore, it includes a *log4j* based logging system and a few bug-fixes from the older version"[16].

Internet can be considered as a distributed, massively parallel supercomputer that is able to connect different elements such as repositories, databases, intelligent agents, and mobile code. It is possible to send your personalized agents to roam Internet in order to perform diverse task such as monitor Web sites.

Talking about Aglets is talking about mobile agents that are a technology that promise many benefits in net work computing. In general terms, "a mobile agent is a program that can migrate from one computer to another for remote execution". There are several characteristics that make a language useful to be use for writing mobile agents and are shown below:

- Their support of agent migration.
- Their support for agent-to-agent communication
- They allow agents to interact with local resources
- Security mechanisms
- Execution efficiency
- Language implementation across multiple platforms
- The language's ease of programming of the tasks mobile agents perform.

For these reasons, Java is an appropriate language for Aglets development.

"Aglets are Java objects that can move from one host in the network to another. Aglet that executes on one host can suddenly halt execution, dispatch to a remote host, and start executing again. They can even take along its program code as well as the states of all the objects they are carrying"[16].

Main objectives of Aglets are [16]:

- "Provide an easy and comprehensive model for programming mobile agents without requiring modifications to Java VM or native code".
- "Support dynamic and powerful communication that enables agents to communicate with unknown agents as well as well-known agents".
- "Design a reusable and extensible architecture".
- "Design a harmonious architecture with existing Web/Java technology".

Finally, it is important to compare Aglets versus Applets because Aglets is considered an evolution of the model of network-mobile code implemented by Java applets. Both technologies require a host Java application to be running on a computer, by the most important difference between Aglets and Applets is that when an Aglet travels across the networks, it carries its state in order to continue running, because an Aglet is a Java running program.

2.5.2.2 Definition

An Aglet (or "agile applet") is a “small application program or applet with the capability to serve as a mobile agent of services in a computer network”. An aglet has these characteristics [16]:

- **Object-passing capability.** “It is a complete program object with its own methods, data states, and travel itinerary that can send other aglets or pass itself along in a network as an entity”.
- **Autonomous.** “An aglet has the ability to decide on its own what actions should be taken and where and when to go”.
- **Interaction with other program objects.** “It can interact locally with other aglets or stationary objects. When necessary, it can dispatch itself or other aglets to remote locations in order to interact with other objects”.
- **Disconnected operation.** “If a computer is currently disconnected from the network, the aglet can schedule itself to move when the computer is reconnected”.
- **Parallel execution**. “Multiple aglets can be dispatched to run concurrently in different computers”.

An aglet is a class or template in the Java object-oriented programming language, and the mobile agent instances of its use are also called aglets.

2.5.2.3 AGLET model structure

The basic elements of the Aglet Model are the following:

- **Aglet:** a mobile java object that visits aglet-enabled hosts in a computer network. It is autonomous and reactive.
- **Proxy:** representative of an aglet that protects the aglet’s public methods like a shield from direct access. A proxy is also able to hide an aglet real location (location transparency).
- **Context:** an aglets workplace. A context provides a uniform environment in a realm of heterogeneous hosts for an aglet to run in.
- **Identifier:** every aglet has a globally unique identifier bound to itself unchangeable during its entire lifetime.

2.5.2.4 AGLETS workbench

The IBM Aglets Workbench (AWB) allows users to create aglets. The AWB consists of a development kit for aglets and a platform for their execution. It is based on the aglet object model, whose major elements are aglets, contexts and messages. The Aglet Transfer Protocol (ATP) and the Aglet API (A-API) are further AWB components that define how to transport aglets and how to interface with them and

contexts. “Both the Agent Transfer Protocol and the Workbench framework protocol have been offered to the Object Management Group (OMG), an industry standards body, as a proposal for a standard Mobile Agent Facility. IBM is offering the Workbench for free to developers”.

Why to choose Aglets?

- One of the pioneer agent technologies using Java.
- Mobility and itinerary characteristics keys are both really important when agent technology is used, and Aglets Workbench support them. It’s enforced with good security facilities: If an Aglet wants to move to a remote host, this one has to be running an aglet server and security measures solved.
- They have some key characteristics to improve agility.

Aglet Structure
The following picture shows us the parts of an aglet: proxy and core. In aglet core we can found all the aglet’s methods and internal variables, and it’

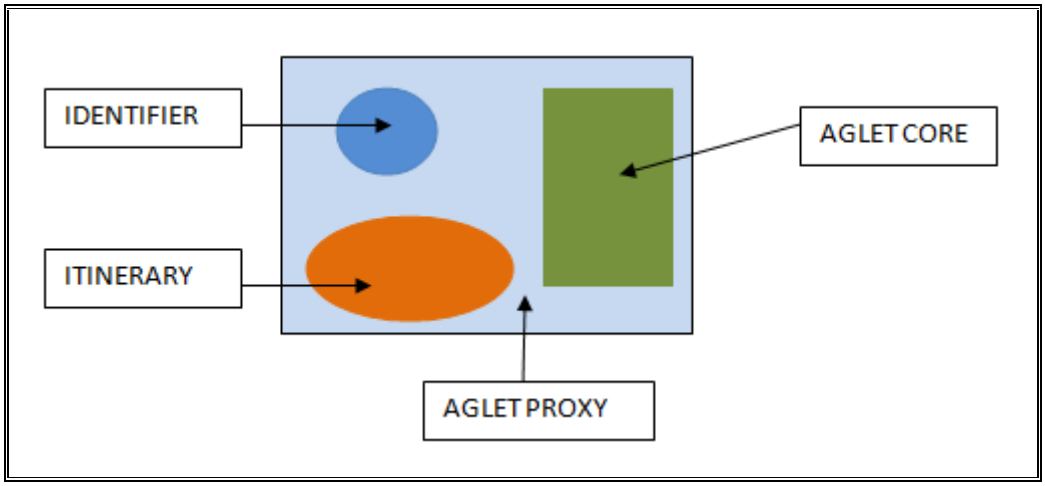


Figure 8 - Structure of an Aglet

2.5.2.5 Decreased Network Load

Network load is reduced using mobile agents because each aglet migrates to a desired local server, then once there, aglets start sending message each other. By the way, message passing could be asynchronous or synchronous.

Aglets mobility facilities:

- **Agent Transfer Protocol (ATP):** Is an application-level protocol for distributed agent-based systems. ATP offers the opportunity to handle mobility of the agents in a general and uniform way, it means it does not matter in which language is programmed each agent when we are transferring mobile agents between computers.
- **Java Agent Transfer and Communication Interface (J-ATCI):** Is a simple and flexible programming interface that enables the developers to work with ATP but without having to build it into them for wire communication.

2.5.2.6 Itinerary class.

Agents have the ability to make decisions by themselves based on their environment, carrying out their tasks, even when the connection with their host is down. These tasks are programmed implanting the **itinerary class** associated to each Aglet, it contains:

- Travel plan.
- Host destination.
- Actions to do at host.

2.5.2.7 Aglets and their environment

There are two important levels of communications present at Aglet environment. We are talking about communication between aglets and hosts, and communication between aglets.
When aglets tries to interact each other, it is carried out using a AgletProxy object on behalf an aglet, even if they are in the same host, and it is the responsible of requesting aglets to take actions. These communications are always established through an agent host, for this reason, it is not very common to invoke some agent's method directly. On the other hand, when an aglet tries to establish a communication with a host, it is carried out using an AgletContext object to get a context identifier and after that adding new aglet.

2.5.2.8 Server Objects: Contexts.

Servers behavior can change without the consent of the owner, this is caused by a mobile agent, it can migrate to a compatible server called contexts, where aglets can interact with each other. This compatible filter is a good security measure, because if an aglet tries to access to a private data the security manager stops the access.

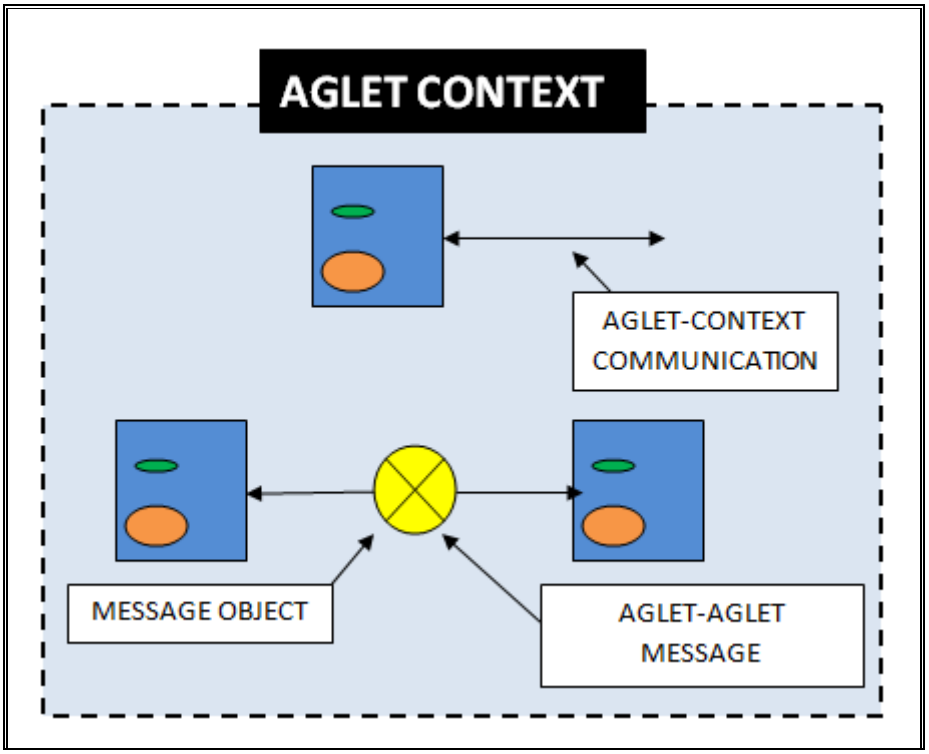


Figure 9 - The Aglet Environment

Aglets could be used for transmitting malicious programs, but in order to prevent these attacks aglets hosts have very severe security restrictions to avoid actions of aglets that has not been originated locally.

2.5.3 COUGAAR (Cognitive Agent Architecture)

2.5.3.1 Introduction

Cougaar is an acronym for "Cognitive Agent Architecture" and it is defined as a Java-based software architecture for building highly scalable distributed agent-based applications in a powerful and maintainable way. In fact, it is designed to support tightly coupled members effectively using a hybrid of shared-memory blackboard interactions, and loosely coupled members using distributed message passing.

Cougaar was a DARPA (Defense Advanced Research Projects Agency) project developed from 1996 until 2001 under Advanced Logistics Program or ALP, whose purpose was to model military logistics using distributed agent technologies. Here we have some points demonstrating how complex the military logistics problem is:

- Millions of different object types to be managed
- Tens of thousands of different interleaved discrete business processes
- Thousands of different organizations with their own physical plants, constraints and user requirements
- Complex, continual interplay between planning and execution
- Over a thousand legacy databases and systems with different data models and protocols.

Over a period starting in 2001 until 2004, Cougaar development and maintenance was carried out, and then continued by BBN under a new DARPA program, Ultra*Log, to enhance the Cougaar platform with components offering: robustness, security and scalability.

The complexity of managing an ontology of so many distinct object types using standard software tools is really inadequate.

There are two factors determining that Standard software modeling techniques are really inadequate to be used: managing an ontology of a lot of distinct object types and the complexity of developing a standard top-down decomposition of the aggregate business processes. In this way, Cougaar was designed to approach these problems, not only reduced to the military domain, because it is an independent architecture for large scale distributed agent systems. Following categories would be enhanced using Cougar:

- Problem domains that entail hierarchical decomposition and tracking of complex tasks.
- Complex application domains involving integration of distributed separate applications and data sources.
- Domains involving the generation and maintenance of dynamic plans in the face of execution.
- Highly parallel applications with relatively loose-coupling and low-bandwidth communications between parallel streams.
- Domains too complex to model monolithically, best modeled by emergent behavior of components.

2.5.3.2 COUGAAR architecture

Cougaar is a large-scale workflow engine built on a component-based distributed agent architecture. Within agents, components interact via a local publish-and-subscribe mechanism. The agents communicate with one another by a built-in asynchronous message-passing protocol. These relationships are dynamically negotiated, using a hierarchical service discovery mechanism. Agents organize themselves into communities to monitor security conditions and agent availability, allowing them to adaptively control their behavior.

2.5.3.3 COUGAAR society

In the Cougaar architecture, an Agent is the principal element and this autonomous software entity models a particular organization, business process or algorithm. A Cougaar ***Society*** is a collection of Agents that interact to collectively solve a particular problem or class of problems. The problems are typically associated with planning, where the plan objective and constraints may be continually changing and replaning in the face of execution. A Cougaar ***Community*** is a notional concept, referring to a group of Agents with some common functional purpose or organizational commonality. Thus a Cougaar society can be made of one or more logical communities, with some Agents associated with more than one community and other Agents not associated with any. The society shares a DNS-like ***Namespace*** that allows all agents to resolve references to one another, and which may be monolithic or distributed/redundant. Within a Cougaar Society, agents execute on a *node,* which is a single Java Virtual Machine containing multiple agents. All agents on the same node share the same processor, memory pool, disk and communication channels. The allocation of agents to nodes is not necessarily domain related, but rather based on a distribution of agents to available resources.

2.5.3.4 COUGAAR Community

A Cougaar community is a notional concept. Things look similar to the view of a society: it is composed of other communities, perhaps sub-communities, and Agents with some common purpose or organizational relationship. Communities are often hierarchical and a given Agent may be a member of several communities.

The most significant characteristic of a community is that it presents a single coherent interface to the rest of the society for a given capability. Ideally, we can identify such interfaces that encapsulate and hide the internal structure of how a given function is accomplished. Recursively, as this implementation breaks down into modules, it makes sense to think of these sub-components as sub-communities, each one with its own interface.

A Cougaar application using communities will typically group agents with similar roles, physical attributes and/or organizational features. Members of a community may also be assigned particular roles therein.
Attribute-based messages facilitate the delivery of messages in a community or between communities. In this way, there is some notional interface of what it provides to the society. There is typically more traffic among Agents in a community, using terms and activities only this community understands, compare to traffic between communities.

In Figure 10, we see a Cougaar community containing a series of logically related Agents.

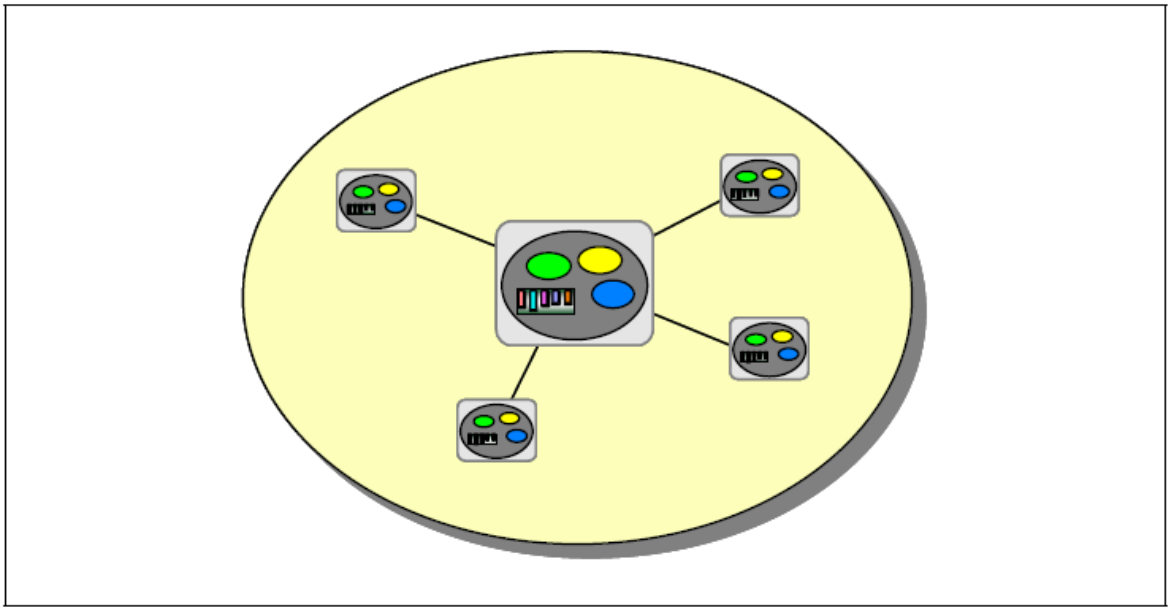


Figure 10 - Cougaar Community

2.5.3.5 COUGAAR Node

A ***Node*** is a single Java Virtual Machine (JVM) instance that may contain and maintain multiple Agents. Talking about efficiency, it is recommended a 1:1 correspondence between the node and the hardware platform. Agents are grouped into a node based on equitable sustainable sharing of computer resource requirements among all agents. Therefore it is helpful to think of a node as a special class of unnamed Cougaar communities, where the logical grouping is by physical machine locality.

Agents, belonging to the same node, are sharing the same CPU (memory and disk) competing for bandwidth traffic. The node works like a router of messages for the Agents it is hosting: messages to other Agents in the same node are passed directly within the same JVM, efficiently short-circuiting the message transport layer; messages to Agents in different nodes pass through a Message Transport layer, that passes the message through the network to the receiving node, which then routes it to the appropriate Agent within that node.

In Figure 11, we see a Cougaar node containing several Agents, with message traffic passing among internal Agents as well as inter-node message traffic for external Agents.

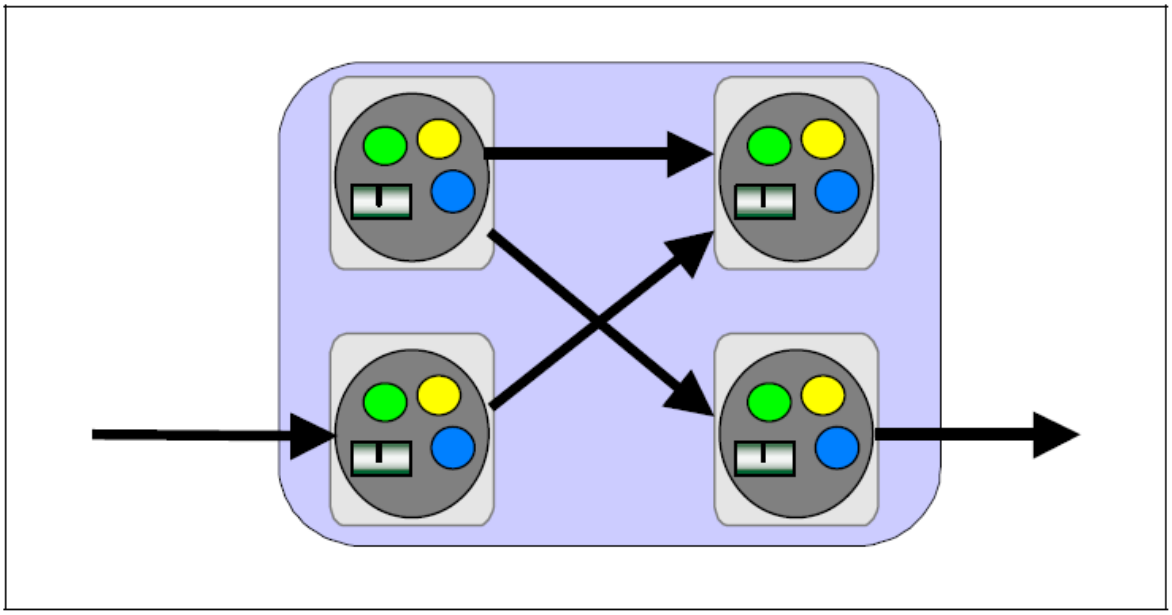


Figure 11 - Cougaar Node

2.5.3.6 Cougaar Agent Internals

Agents are the first-class members of a Cougaar society. Agents communicate with each other point-to-point by sending messages via the node-level Message Transport facilities. Agents are themselves completely generic: all Agents in a society will, for most purposes, have an identical code base and being instances of the same class.

At the most basic level, Agents contains a Blackboard and one or more Plugins. The Plugins are software components that are added to an agent to contribute with a specific piece of application business logic. These Plugins add a domain-specific behavior which determines how the agent responds to the messages received from its peers, and they operate by publishing content and subscribing objects via a fairly traditional-looking Blackboard. These Cougaar Blackboards are not themselves distributed, and its modifications are transaction controlled using a membership model, but it never changes to referenced or inner structure. Blackboard contents are segmented into sets of logically-related objects by Domains.

A Domain is a specification of the language used by Plugins to communicate with each other and with related Plugins in other Agents. Each Domain contains several LogicProviders which are components that transform blackboard objects into messages to be properly received into other Domains. These messages are queued and managed by each Agent.

Figure 12 illustrates aCougaar Agent with its internal structure exposed, showing the Blackboard and Plugins.

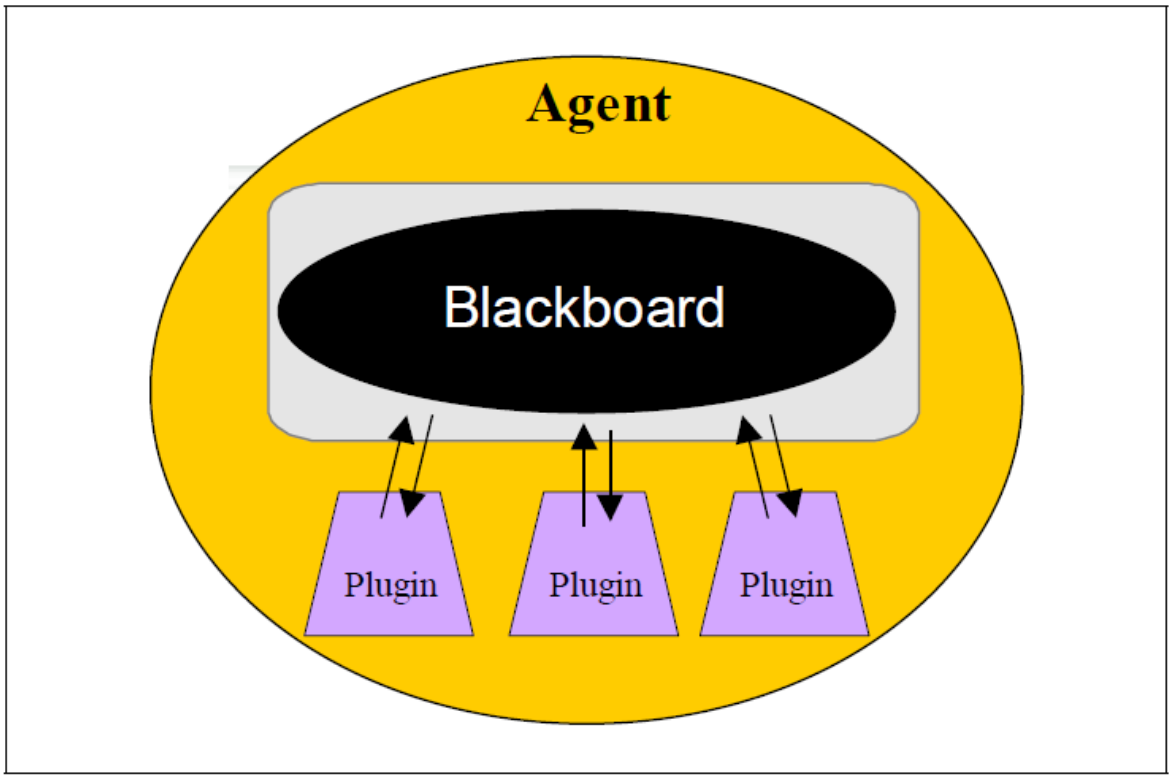


Figure 12 - Agent Internal Structure

2.5.3.7 Cougaar Blackboard Contents

Firstly, Plugins publish and subscribe any object to the Blackboard, but the only necessary object types needed for collaborative planning are Tasks, Assets, and PlanElements. Tasks are defined like requests from one agent to another to plan an operation. When tasks are created or decomposed they eventually must be allocated to an asset, in this way, assets are simply the consumers of tasks. For example, an important type of asset is an Organizational Asset, which represents a proxy to another Agent. Finally, tasks may be allocated to an asset resource, expanded into subtasks or aggregated with other tasks, therefore, disposition of tasks is contained in PlanElements.

Each Agent owns its blackboard and its contents are visible only to that Agent. Every sharing of blackboard state is done by explicit push-and-pull of data through inter-Agent tasking and querying. In this way, Cougaar is able to maintain fine-grained state in individual Agents while sharing only high-level synopsis information around the society, making the management of information scalable and efficient.

There are two fundamental concepts related to distributed management of the blackboards:

- **“Time-phased locality of reference”:** Representation of an object may be present parallel in multiple Agents without fear of conflict, but it is only actively managed at one place at time.
- **“Managed inconsistency”:** Agents work independently and asynchronously on messages passed from one to another. They respond dependently and asynchronously to replies received from Agents, so there is no synchronization control imposed. Despite of everything, this feature allows the agent to work without delays waiting for another Agent to get its job done, or

recover from a network problem, or a burst of requirements from yet another agent. Although, it clearly shows the explicit inconsistency of Cougar.

Figure 13 illustrates the contents of the blackboard of a given Agent. The blackboard contains a dynamic chain of plan elements, tasks and assets.

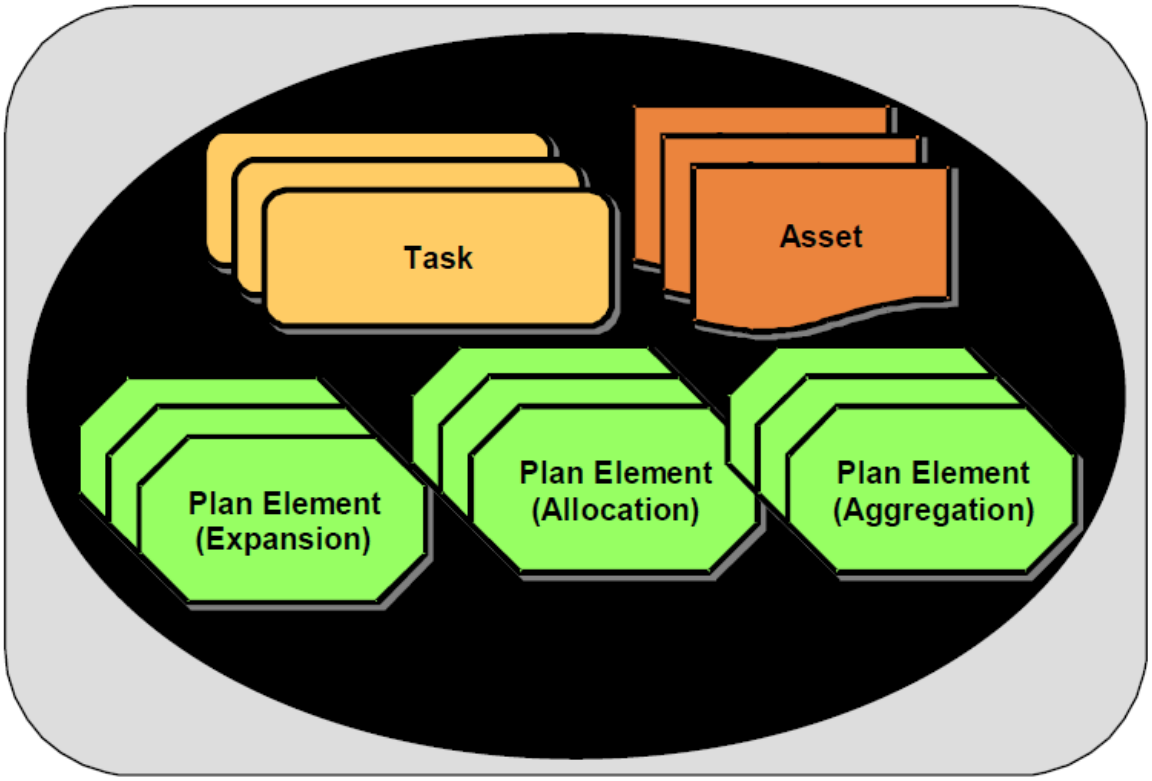


Figure 13 - Agent Blackboard Contents

2.5.3.8 Top Level Design Principles

Cougar architecture is based on different design principles that can be described as follows:

- **Composability**. This pattern design is very common. It tries to decompose complex problems into smaller. We can see this feature in the components we have discussed:
  - *Agents* are made up of many *Plugin* components, each providing a small piece of business logic or functionality, allowing the Agent's behavior to emerge from the composed pieces.
  - *Assets* are made up of *PropertyGroups*, which contain a cohesive set of data slots, often from a single data source. Assets are further made up of *Prototypes,* which capture standard patterns of PropertyGroups that are invariant over a large set of instances.
  - *Communities* are made up of smaller Communities and *Agents*. Societies are, in turn, made up of Communities.
  - A *Blackboard* contains logically separated sets of objects grouped into Domains by their common application language. Each Domain specifies a series of *Logic Providers*, which allow the Domain's logic to be translated into other co-resident languages, including Cougaar Messaging.

- **Information Hiding and Encapsulation**. Components of the Cougar architecture have access to all the data needed, but no more, enhancing scalability pattern of the Cougar society. It is easy to understand this point by studying some component's behavior. For example, Agents are not allowed to see into the blackboards of other Agents and must use standard interfaces to communicate with other Agents. Plugins cannot see the state of other Plugins. Communities have clear interfaces with one to another, but they hide their internal Agent configurations.
- **Time-Phasing.** Expected quantities, costs, values of different entities will change over the course of a planned operation. All information about physical entities is time-phased.
- **Dynamic Replanning and Execution Monitoring**. A Plan is designed on a continual dynamic negotiation between Agents and Plugins, based on real world requirements, situation information and asset availability. Cougaar forces preplanning when some variables have changed to generate an optimized solution. For this reason, the plan is always monitored.
- **Security**. Infrastructure core software, the Plugin modules and configuration information are designed to be secure traffic interceptions or corrupted configurations.
- **Robustness**. Cougar applications are designed to survive the loss of any individual components and automatic recovery of lost agents. In addition, there are automatic recoveries of lost agents or mechanism to conserve resources and to use redundancies. As it has been already mentioned, an Agent state can be restored or restarted, or in other cases, it can be waiting for another agent due to agent failure or network outage.
- **Scalability**. Several actions have been adopted to design and maintain a scalable Cougar architecture. By encouraging encapsulation, data hiding, and fine grained information management, the information passed between Agents is limited to a bare minimum. By establishing peer-to-peer inter-Agent communications, exponential growth in the interdependencies and interactions among different agents is avoided.

2.5.3.9 Top-Level Information Flow Concepts

Giving a general view, a Cougaar society is started at some point by hand or by an automated Application Server process. Nodes can be manually killed or restarted, while new Nodes can join the society dynamically. Agents belonging to these Nodes start to interact with one another, databases, legacy systems and sensors to fill their Agent with Assets. Due many stimuli (real-world data from databases or sensors) Cougaar expects to be continually processing, continually trying to find a better solution to the given problem and continually reacting to changes in resources, in requirements and in events monitored from execution.

Tasks are decomposed (by Expansion) and assigned (by Allocation) to other processing units, either in the same Agent or in another Agent. Each task, then, creates a "channel" for information flowing through the society for requirements passing down, and responses flowing back up. At each point, the execution of the planned requirements is monitored, and replanning may occur if significant discrepancy is detected between the planned operations and the observed ones.
However, through this flow of information up and down processing chain, there are many negotiations among different Plugins and Agents to perform a more global optimization over a larger space. By changing task allocation and task preference, an Agent can search for an optimal solution between Agents, or manage relationships with multiple providers to optimally satisfy aggregate requirements.

A Cougaar application has two equally-important levels of communication present and active at any given time:

- **Agent-Agent,** where agents communicate with each other as peers, hiding the internal business logic and allowing loosely-coupled, asynchronous and widely distributed problem solving, perhaps with Agents located with their supporting external systems (data sources, humans, etc). Relationships between agents need to be unique, time-phased and dynamic. At the same time, different roles like superior/subordinate or customer/provider are shown.
- **Plugin-Plugin,** where components communicate with each other through the Agent's Blackboard using tightly-coupled, transactionally-protected interactions. Plugins are often vastly different from each other, based on the task performed by each one.

At the same time, Cougaar includes support for distributed agent naming services. These naming services are used by the Cougaar message transport to route message over multiple network protocols to mobile agents. Application developers can also use the naming services to dynamically discover agents at runtime.

Following, the five different types of distributed naming capabilities that have been identified are presented:

- **"Name Generation"** constructs a globally unique agent name.
- **The "White Pages"** is a table that maps names to network addresses (e.g. DNS).
- **The "Yellow Pages"** is an attribute-based directory (e.g. a categorized phone book).
- **"Local Discovery"** uses LAN-based IP multicast to locate nearby agents.
- **"Peer-to-Peer Search"** allows an agent to search adjacent agents for resources.

2.6 ADVANTAGES AND DISADVANTAGES (JADE VS AGLETS & COUGAAR)

Discussing general advantages and disadvantages of Jade versus Aglets and Cougaar, it can be concluded that Jade is the best technology to be used.

Parameter	Best technology	Reason
Developer tools	Jade	Available easy monitoring and debugging tools
Large-scale design	Cougaar	Best suited to developers that want to customize the agent framework's core services or create complex, large-scale, robust or highly secure agent-based applications
Messaging	Jade	Communication messages formatted in ACL or XML. Pluggable transport protocols include RMI, IIOP and HTTP
Mobility	Aglets	Enhance Agents mobility
Resources available	Jade	Updated and easier to find support
Security	Jade	It is built as a plug-in
Simplicity	Jade	Best suited from the developers' point of view

Share information between different agents problem	Cougaar	Provides a nice solution for this problem
Support and Documentation	Jade	Active and updated
Updated	Jade	Active support and forum community to report bugs

Table 2 Best technology selected by parameter

2.6.1 Jade advantages

- **Messaging**: In Jade agent platform, inter-agent communication messages are formatted in ACL or XML, and pluggable transport protocols include RMI, IIOP and HTTP. Java utility classes simplify the construction and handling of FIPA-compliant ACL messages. They are the most extended and ideal for low-volume agent interactions that require cross-platform interoperability. On the other hand, Aglets is MASIF compliant and Cougaar is not standard compliant.
- **Updated**: Talking about updated documentation, active support and forum community to report bugs, it is important to mention that Jade takes a great advantage versus the others.
- **Simplicity**: Jade is the best suited from the developers' point of view, thanks of its simplicity design.
- **Developer tools**: Available easy monitoring and debugging tools.
- **Support and Documentation**: Active and updated.
- **Resources available** (libraries, APIs, code...): updated and easier to find support than the others.
- **Security**: In Jade it is built as a plug-in. It provides a security model based on principals, resources and permissions, which enables authentication and authorization of both agents and the owners.

2.6.2 Jade disadvantages

- **Large-scale design**: Cougaar is best suited to developers that want to customize the agent framework's core services or create complex, large-scale, robust or highly secure agent-based applications. But perhaps cost-benefit analysis is not appropriate for other applications.
- **Share information between different agents problem**: For example, Cougaar's blackboard provides a nice solution for this problem, for this reason it is a good alternative to combine Jade and Cougaar.
- **Mobility**: it is not a key element in JADE. It focuses on other functionalities relevant to the development of multi-agent systems. The JADE built-in agent Mobility Service supports mobility among containers within the same JADE platform. On the other hand, Aglets is directly design to enhance Agents mobility.

3 SOA LIFE CYCLE

3.1 SOA LIFE CYCLE PROPOSALS

3.1.1 Dan Foody Proposal

Dan Foody has done a life cycle proposal as SOA affects the traditional SDLC (Software development Lifecycle).

In general terms SDLC has two phases:

- **Pre-production:** This phase includes the design, development, QA ... Everything that happens before taking the software production.
- **Production:** This phase include the deployment, operation…Everything that happens after taking the software production.

In a SOA lifecycle, Dan Foody proposes a new phase between Pre-production and Production called Pre-consumption. The new phase is a hybrid of both phases. This means, part of the build is in production and part is in pre-production. This is explained below:

- Pre-consumption is looked like part of production phase by service providers because services are completed and operating in production.

- Pre-consumption is looked like part of pre-production by service consumers because consumer application is still unbuilt.

3.1.2 Miko Matsumura Proposal

Miko Matsumura proposal states that the SOA lifecycle is different from the traditional SDLC. The proposal divides the lifecycle into three phases:

- **Design time:** In this phase a business application is formed putting the services together.
- **Runtime:** In this phase start implementing SOA and the business activity is initiated.
- **Change time:** In this phase the inevitable alterations and business requirements changes occur to provide agility.

The particularity of the proposed life cycle is that it begins where the traditional SDLC ends. The definition of design time covers the assembly of the now published services into a business application.

In this proposal the policies also have a lifecycle. In this sense the policies have a design time, runtime and change time, because the constraint model must be adaptable, flexible for compliance agile SOA promise.

3.1.3 IBM Proposal

IBM proposal introduces the role of SOA Quality Management in SOA Service Lifecycle. The IBM SOA lifecycle has four phases: Model, Assemble, Deploy and Manage.

- **Model:** Capture the business design requirements and objectives.
- **Assemble:** Convert the design business into the definition of business processes. This phase automates the integration of business process services for service developers assemble reusable assets that architects have modelled.
- **Deploy:** Create the runtime and deployment environment of business processes. In this phase management functions deploy the services that are tested and released.
- **Manage:** Manage and monitor services and business processes defined.

3.1.3.1 SOA Quality Management

IBM SOA Quality Management provides the following key capabilities:

- Enable tools and best practices focused on quality management.
- Eliminate process redundancies for optimize workflows across business process
- Enable functional and performance testing of business services to ensure business agility.

The following figure shows the Quality management in the lifecycle and then the role of quality in each of the phases of lifecycle is explained.

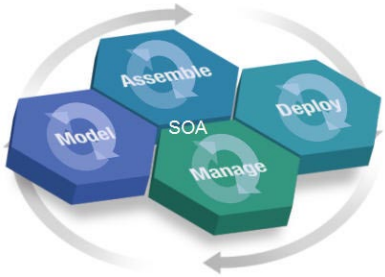


Figure 14 - SOA Quality Management

- **Model:** The aim is to validate that business requirements have been modelled and the design is correct.
- **Assemble:** The aim is to validate that the services created are necessary to compliance the requirements of business.
- **Deploy:** In order to ensure the performance and scalability of service.
- **Manage:** The aim is monitoring and tracking perform service into the inventory of services.

3.2 IBM LIFE CYCLE

The life cycle of IBM SOA Foundation includes the following phases: Model, Assemble, Deploy and Manage.

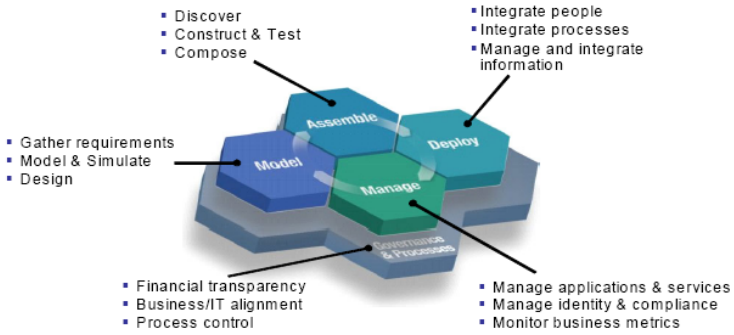


Figure 15 - IBM Life Cycle phases

3.2.1 Model

The process of capturing the business design and understanding the requirements and objectives is known as modelling. This phases aims at gathering requirements, model and design. The business process specification contains the business requirements.

The model also incorporates key performance indicators that are important measurements of the business, such as business metrics.

3.2.2 Assemble

The business design communicates the business objectives to the IT organization, which assemble the information system to implement the design. This phase is aimed at turning the business design in defining business processes. If it is necessary to build new services, these will be built following a service-oriented development.

Finally, the assembly phase includes applying policies and conditions to control the way in which the applications operate in the production runtime environment.

3.2.3 Deploy

This phase includes a combination of creating the hosting environment for the applications and the deployment tasks of those applications. In this sense, deployment phase resolves the resource dependencies of the application, setting the operating conditions, capacity requirements and the integrity and access constraints.

3.2.4 Manage

Manage aims at monitoring the services. This phase include tasks, technology and software necessary to manage and monitor the services deployed in the production runtime environment.

Monitoring is focused on evaluating and following the performance of the service. Management system involves routine maintenance; manage applications, application security and scalability prediction.

3.2.5 SOMA Methodology

Service Oriented Modelling and Architecture (SOMA) tries to add key features to the design and delivery of services. SOMA searches approximate RUP to a service-oriented programming. The SOMA methodology has four phases: Identification, Specification, Realization and Deployment. The following figure illustrates the SOMA methodology.

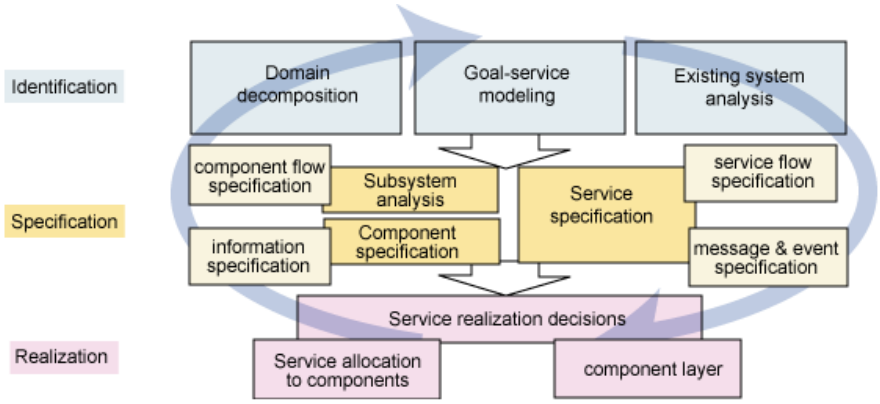


Figure 16 - SOMA Methodology

3.2.5.1 Identification

This phase aligns the business with information technology. There are no differences between the process of developing new services and the previously functional services identification. It must start from a business model where services should specify its needs in order to be implemented as well as a list of functional requirements that must be covered. On the other side, if necessary, non-functional requirements can be added through KPI (Key Performance Indicator). The input artefacts are the business model and functional requirements, while output artefacts are the initial service model and operations of services.

Below a brief description of the tasks contained in this phase is explained:

- Identify services from the objectives: The objective is to identify all functional services that make up the system to later link them to functional requirements that will be supported. The result of the above is a model of technological services associated with each functional service. It is necessary to identify the consumers of services, business process, subprocesses, activities and tasks. That is how it determines which operations will belong to the service specification, and the parameters of these operations.
- Develop an analysis of existing assets: The aim is to determine what currently exists and can be reused.

3.2.5.2 Specification

The main objective of this phase is to completely specify SOA design elements from the architectural point of view. Service specification can be viewed as designing your own service model. This phase provides an architecture for services so that all customers, suppliers, specifications and partitioning are specified in terms of its structure and behaviour. The output artefacts are the service model, service interfaces and service messages.

Below a brief description of the tasks contained in this phase:

- **Structure service architecture:** The goal is to build the service model to construct the interactions.
- **Validate and classify services:** SOMA provides a tool called Service Litmus Test, which serves to identify which services should be published and ensures that services are aligned with a requirement, which in turn promotes the search and reuse of services.
- **Identify service partitions:** The goal is to take the service model of each Functional Service and Technology Services that will identify each of the partitions according to the following structure: services consumers, business application services composite, business application services atomic and infrastructure services.
- **Model atomic service providers**: Identify the business application services and atomic services infrastructure.
- **Model composite service providers:** Identify service providers that will result from the composition of several services
- **Model service consumers:** Specifications of the services consumer of business components.
- **Assign the services defined in the service partition**: After modelling the different types of services, they should be put together in the same model and the relationships between them should be established.
- **Consider service policies:** This refers to the non functional requirements that must be covered by the system. This includes details like the performance, capacity, availability and security.
- **Refine service architecture:** Collect the work done in the previous task and finish it completely specifying the structure and behaviour of the architecture of functional services. This task designs the interaction and the collaboration between services. Through the relationships between services, consumer services can be validated.
- **Design parameter types, messages and information on the types:** This refers to specify types of design operating parameters, messages and information.
- **Validate the final model of service**: It is necessary to ensure that services can be deployed independently and that they have all the information needed to implement.
- **Model the service assembly:** This task aim at completing the development model of the service with pieces of software.
- **Model the service delivery:** Model the deployment infrastructure in terms of both physical and logical nodes.

3.2.5.3 Realization

Up to this phase, it is known what the service does, what it expects and the result of the invocation. The output artefacts are the class diagram of the services and the interaction diagram for each service.

Below a brief description of the tasks contained in this phase:

- **Creating the Model Structure:** This is an iterative process that starts creating the structure of the model. For each component of the service a package UML will be created. A component of service is the realization of the specification of it. This include structure, behaviour and polities contract.
- **Creating service components:** Service components are created from the service model.
- **Refine service components:** Once service component is created, it is refined and deepen, following the guidelines.
- **Creating classes:**
    - A facade class designed to directly carry out the implementation of the specification. This class is called *ServiceNameServiceFacade.*
    - An implementation class designed to perform the detailed structure of the service component and its tasks. This class is called *ServiceNameServiceImpl.*
    - An interface representing an embodiment of the service specification. The interface is called *ServiceNameService*.
- **Refine the interface:** Add to the interface the operations defined in the service specification.
- **Apply pattern design**: Firstly, applying the Façade pattern, which needs that the façade has the same operations to the interface. Second applying the Interface pattern, which need that the implementation class extends of the interface.
- **Using the specification of asset reuse:** The organizations have an internal repository of patterns, frameworks and best practices. This task links the asset with the realization of the service.
- **Designing the class structure:** It conducts a class model for each of the identified services.
- **Designing the behaviour of classes:** It conducts the interaction diagrams between the classes that make up each service.

4 CONCLUSIONS

Once the analysis of the different architectures have been carried out, it has been decided that the Service Oriented Architecture adapts better to the technical goal REMOTE project pretends. As the project is not so based in the implementation of network systems or in hardware development, the best choice has been SoA due to its better adaptation to the needed characteristics of the project related with the Ambient Intelligence.

As SoA supports the usage of multi-agents, several options have been contemplated. Taking into account the advantages and disadvantages of every option, as well as the comparison between them, finally the agent architecture JADE was selected.

Regarding the several lifecycles that can be used and have been analyzed within the SoA architecture, the final one to be use would be the IBM proposal. A more exhaustive study has been done in order to justify its usage within the project. Due to the IBM lifecycle the methodology linked to it is the SOMA Methodology that will help in the modeling and development of the REMOTE's final architecture.
The definition and follow of the different phases of this methodology, would result in the final REMOTE platform services.

REFERENCES

[1] “Service-Oriented Architecture: Concepts, Technology, and Design”, Thomas Earl, Prentice Hall 2005.

[2] “SOA: Principles of Service Design”, Thomas Earl, Prentice Hall 2007.

[3] P. Maes. Modeling adaptive autonomous agents. In C. G. Langton, editor, Artificial Life, An Overview, Cambridge, Massachussets, 1995. MIT Press.

[4] [Dan Foody, “What is the SOA Lifecycle?” Progress Software blogosfera http://blogs.progress.com/soa_infrastructure/2007/11/what-is-the-soa.html, last access 18/11/2009.

[5] John Ganci, Amit Acharya, Jonathan Adams, Paula Diaz de Eusebio, Gurdeep Rahi, Diane Strachan, Kanako Utsumi, Noritoshi Washio. IBM RedBooks,“Patterns: SOA Foundation Service Creation Scenario”, September 2006.

[6] [Ueli Wahli, Lee Ackerman, Alessandro Di Bari, Gregory, Hodgkinson, Anthony Kesterton, Laura Olson, Bertrand Portier. IBM RedBooks, “Building SOA Solutions Using the Rational SDP”, April 2007.

[7] Gary McBride, “The Role of SOA Quality Management in SOA Service Lifecycle Management”.http://www.ibm.com/developerworks/rational/library/mar07/mcbride/, last access 18/11/2009.

[8] http://www.OSGI.org/OSGI_technology/

[9] Rich Steely, “SOA lifecycle: What are we talking about?” http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1213362,00.html, last access 18/11/2009.

[10] UPnP-Forum, “UPnP™ Device Architecture 1.1”, 2008.

[11] “WhitePaperJADEEXP”, F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, September 2003

[12] http://jade.tilab.com

[13] “LEAP user guide”, Giovanni Caire, Federico Pieri, 2003,

[14] “Developing multi-agent systems with JADE”, Wiley, 2007

[15] B. Sommers, “Agents: Not Just for Bond Anymore,” *JavaWorld*, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-agents.html April 1997.

[16] B. Venners, “The Architecture of Aglets,” *JavaWorld*, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html, April 1997.

[16] Aglets development group. “Aglets 2.0.2 User’s Manual”. March, 2002.

[18] Rahul Jha and Sridhar Iyer. “Performance Evaluation of Mobile Agents for E-Commerce Applications”. International Conference on High Performance Computing (HiPC) , Hyderabad, India, Dec 2001.

[19] Programming and Deploying Java™ Mobile Agents with Aglets, Danny B. Lange/Mitsuru Oshima, Second Printing, Addison Wesley 1998.

[20] Aglets Open Source web site, http://aglets.sourceforge.net/

[21] Cougaar Open Source web site, http://www.cougaar.org

[22] Aaron Helsinger, Todd Wright. BBN Technologies. "Cougaar: A Robust Configurable Multi Agent Platform"

[23] Denis Gra˘canin, H. Lally Singh, Michael G. Hinchey, Mohamed Eltoweissy, Shawn A. Bohner. "A CSP-Based Agent Modeling Framework for the Cougaar Agent-Based Architecture". IEEE Computer Society ,2005.

[24] Cougaar architecture document. Technical report, BBN Technologies, 5 July 2004. Version for Cougaar 11.2.

[25] Ronald D. Snyder, Dr. Douglas C. MacKenzie. *Mobile Intelligence Corporation. "*Cougaar Agent Communities*".* Appears in Proceedings, Open Cougaar 2004, New York, 2004.

[26] Ronald D. Snyder, Dr. Douglas C. MacKenzie. *Mobile Intelligence Corporation. "*Robustness Infrastructure for Multi-Agent Systems*".* Appears in Proceedings, Open Cougaar 2004, New York, 2004.

[27] William J. Tolone,David Wilson, Anita Raja, Wei-Ning Xiang, E. Wray Johnson. "Applying Cougaar to Integrated Critical Infrastructure Modeling and Simulation", 2003.