ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria elettronica, informatica e telecomunicazioni

AGENTS, NODES & RESOURCES: UNIVERSAL NAMING SYSTEM FOR A COORDINATION MIDDLEWARE

Elaborata nel corso di: Sistemi Distribuiti

Relatore:
Prof. ANDREA OMICINI

Co-relatori:
ING. STEFANO MARIANI

Tesi di Laurea di:
FEDERICO FOSCHINI

ANNO ACCADEMICO 2012–2013
SESSIONE II

PAROLE CHIAVE

TuCSoN

Cassandra

Naming

Locator

Research

”Don’t waste time living someone else’s life.
Stay hungry. Stay Foolish.”

Contents

Introduction

The thesis aims at design a model, architecture, and technology for the Naming System of the TuCSoN Coordination Middleware, including agents, nodes, and resources. Universal identities accounting for both physical and virtual mobility will be defined, moving towards a comprehensive, distributed Management System dealing also with ACC and transducers, foreseeing issues such as fault tolerance, persistence, consistency, along with disembodied coordination in the cloud.

During the design of this Naming System we have decided to design also a new Locator System in order to upgrade all functionalities about the Naming and Locator System in TuCSoN . Will will also introduce a new Research System in order to give to any consumer entity, a complete Research Service that provides all data about the Naming and the Locator System. Starting from this defined parts of the module that will be explained, we have decided to give to this module a name: **LoReNa**; this is an acronym standing for **Locator**, **Research** and **Naming**.

First of all we will introduce and define all the background knowledge useful to understand all details about the system that has been designed; in fact, the first chapter is dedicated to the TuCSoN description and all its features in terms of model, architecture and technology with the description of the concept of Tuple Centre and Tuple Space because these are very important parts of that coordinated middleware.

Another foundamental part of the background knowledge is the description and definition of Cassandra because this is the data store distributed model adopted in the designed system in the layer about the data store.

After the background knowledge chapter we will describe all details about **JADE**. This is a Java based framework for *interoperable*, *intelligent*, *multi-agent system*. Like the other background knowledge, also for this framework, all details about its model, architecture and technology will be explained. This framework is foundamental because its model is an important factor for the development of the designed model that will be described; in fact, the **LoReNa**'s model has been designed basing on the **JADE**'s one but, in the model of **LoReNa** we have introduced other functionalities such as the Locator module. Also the architecture of **LoReNa** has been studied starting from the **JADE** one but it's more oriented to the network distribution than the other one because it's designed according to the previously cited **LoReNa**'s model. Dealing with the fact that TuCSoN is *Java* based, we have decided to give to **LoReNa** a *Java* base but, as will be described in the relative chapter, this module can be adopted with any programming language.

In the third chapter of this thesis the **LoReNa**'s *structure* and *interaction model* will be detailed described in all its parts, starting from the Naming Service Model with all its details about the three layer structure. Also for the Locator Service Model there is the complete description about the same three layers because these two models are very similar in terms of structure and interaction. The unique different model is the Research Service one because it communicate with the others in order to sadisfy all research requests and it's divided in two layers.

In the next chapter the new Naming Syntax, described as Universal Naming Syntax, is introduced and described in all its details, starting from its definition. In this chapter also the procedure that an entity has to do is explained in order to give a complete description of all aspects of this syntax.

In the fifth chapter all the case studies will be introduced in all details in order to give all aspects in terms of computation in all cases, both as local and distributed one for all services contained in **LoReNa** (so, these are Naming, Locator and Research services). In the second part of this chapter all details about the Fault Tolerance Case Studies will be described in order to give a complete useful description of all policies and procedures relating to solve all faults, errors and issues.

Logically, it's impossible to treat all fault cases, so we have decided to explain all details about the Fault Tolerance relating to the foundamental issues (for example relating to the crash of internal entities or the connection unavailability, etc.) in order to provides a useful report of all relative procedures of these main Fault Tolerance Cases.

The complete description of the main possibly future developments are contained in the next chapter, the sixth one, in order to explain all future development ways for **LoReNa** because this is an important aspect for a new system as this one.

The last chapter of this thesis is dedicated to the conclusions of the design and modelling phases in order to give a final and exhaustive report for **LoReNa**. In this chapter will be described all aspects of this system from different viewpoints so as to provide a complete vision of all positive aspects about this module.

Chapter 1

Background knowledge

1.1 What is TuCSoN ?

TuCSoN (*Tuple Centres Spread over the Network*) is a general-purpose multi-agent systems infrastructure (*MAS*) that allows the communication and coordination between agents by making use of **centers of tuples**, which are defined as programmable *tuple spaces*, shared and responsive, in which the Agents can enter, read or consume *tuples*.

1.2 Tuple Centre

In *multi-agent technology* has become crucial the role of coordination to allow heterogeneous mobile *agents* to work collectively in order to "unify separated activities in a set" [1]. Agents are defined as independent software components, possibly distributed and concurrent, based on the goal. Therefore agents are active entities organized in societies and inserted into an environment. The agent systems differs from the object system because they are not based on the controls, but on the goal; in fact in the object system the controls is outside of them, and humans work as a kind of central control authority, while in agent system, the control is inside the system and agents' goal is to driving control [2].

As objects of the OOP paradigm (*Object Oriented Paradigm*), agents encapsulate a *state* and a *behavior*. Unlike objects, agents have control over both, while objects do not have control over their behavior: to send a message to an object causes the invocation and execution of a method. The agents, however, do not interact via the invocation of methods, which causes the shift of control from one object to another, but through some interaction model / coordination, which can be based on messages passing or other abstractions communication / coordination abstractions, such as blackboard, tuple spaces, etc.

Generally speaking [3], we can define a **coordinated system** as a collection of *coordinables* that live and interact in a *coordination space*. In a software enviroment, coordinated entities are indipendent computational elements whose mutual interaction is the global behaviour of the coordinated system.

The coordination limits the interaction between communicating software *components* [4]. Thus, the coordination model is modeling the space of interaction between components, providing the mechanisms, protocols and abstractions used by components to communicate. More precisely, the specification of a model of coordination should include:

- The **coordination space**: the collection of interacting entities and the space of their interactions. More precisely, it defines:
  - What is a *coordinated eligible entity* (a component, an agent, a process, an object, an application, etc.)
  - What is a *means of coordination* (a channel of communication, a connector, a blackboard, a space of tuples, etc.)
  - What is a *communication event*, incoming or outgoing

An *eligible coordinated entity* is any computational entity that sees and realizes both the communication language and coordination language, and from the viewpoint of the coordination space, an *entity* is characterized by its observable behavior, defined by generated communication events.

- The **communication language**: syntax used by the coordinating entities to express the information exchanged between them.

- The **coordination language**: syntax used by the coordinating entities for communicating operations, and its semantics in terms of kind of communication events (incoming or outgoing).

- The **means of coordination**: abstraction that govern the interaction between coordinating entities. This is characterized by an observable behavior (in terms of events in the input or output) and an execution model, which sets the basic rules for the interaction of the components.

  Being *interactive machines* [5], then taking incoming communication events and generating new outbound events, means of coordination are suitable to be described as *interactive transitions systems*, where the communication state is the state of the system. Some transitions are triggered by interaction events, and some transitions generate output events.

  While on the one hand, this approach makes it possible to work with unfinished computations, on the other hand should keep the semantic specification simple, and make it efficient in guiding the implementation process, considering a system of transitions such as a operational characterization that can be directly mapped into the structures of any programming language.

In the specific case of agent systems we are dealing with:

- the *eligible coordinating entities* are *agents*
- the *means of coordination* are *Tuple Centres*
- the *communication language* is the set of *tuples* and *tuple templates* eligible in addition to the *unification function* that connects the two sets
- the *coordination language* is given by the *coordination primitives* executable on the tuple centres
- *communication events* are the execution of the coordination primitives or environmental events associated with the infrastructure (time passing, external resources, etc.).

All these concepts will be discussed and described in detail in the following sections.

1.2.1 The tuple based coordination model

In general, *coordination* is responsible for management of the interaction between components [6], then in the context of a multiagent system it will be directed to the problem of how agents interact. A *model of coordination for multi-agent systems* [7] constitutes a structure with the goal of modelling the interaction space of companies composed by agents (*agent societies*) [8].

The **tuple** based coordination models [3] were initially used in the field of parallel programming, but their characteristics make them ideal for addressing the problem of coordination in open, distributed and heterogeneous systems such as *Internet agents* [9].

In tuple based models, agents interact by exchanging *tuples*, which are ordered collections of elements that contain information. The agents communicate, synchronize and cooperate through **tuple spaces**, inserting, reading and consuming tuples. The main benefits of these models based on tuples are:

- **Clear separation between computation and coordination** [1] (architecture neater)
- **Generative communication** [10] (the generated information has a live time independent from the creation time)
- **Associative access to the interaction space** [11] (the access is based only on the structure and content of information exchanged)
- **Suspensive semantics**

The main features of *generative communication* are forms of spatial and temporal decoupling, based on the fact that anyone who sends and receives informations doesn't need to know each other in a reciprocal way and they doesn't need to coexist in the same space or at the same time to communicate (in this case for exchange tuples).

The *associative access* based on the unification between tuples promotes synchronization based on the content and structure of tuples; the coordination is information driven and allows the knowledge-based pattern ordering implementation.

Finally, the *semantic suspension* allows pattern of coordination based on the information availability, so an agent can suspend itself waiting for a

tuple in the tuple space.
However, the tuple spaces do not provide flexibility and control necessary for complex multi-agent systems development, it's necessary to take a step forward towards the *Tuple Centre* to try to overcome the tuple spaces' limits. A **Tuple Centre** is a space of tuples whose behavior can be defined in terms of *reactions to communication events.* To understand the Tuple Centres functioning we have to start from the study of tuple spaces, which are the basis.

1.2.2 Tuple Spaces

A **tuple space** is an implementation of the *associative* memory paradigm for parallel and distributed programming, so the access to data (tuples) is based on *tuples*'s content and type, not to their physical location [19].

It consists of a repository of **tuples**, defined as *a list of fixed size heterogeneous elements which can be accessed concurrently.* As an idea you can consider the example of producers and consumers, thinking about a set of processors that produce data and a set of processors that consume these data [20]. The producers put their data in the form of tuples in tuple spaces, while consumers consume these tuples taking them from the space of tuples in the case where these match a certain pattern. You can think about tuple spaces as a form of distributed shared memory [19]. In fact, any processor can refer to any tuple, regardless of its specific physical location: even if you have the perception that you are working with a shared memory is not necessary that there is a physical memory shared.

The concept of tuple space had as a pioneer David Gelernter, a professor at Yale University. The first concurrent tuple space based model is *LindaTupleSpace*, it is a coordination language to express the parallel computation and its strength is the ability to describe parallel algorithms without making any specific reference to an architecture [10].

Nowadays there are various tuple spaces based system wich have been implemented for different programming languages.

The tuple spaces functioning is based on five basic fundamental primitives:

- *out()*: insertion of a tuple (asynchronous)
- *rd()*: blocking read of a tuple, if the tuple is not present the process is

suspended waiting for a tuple that satisfies the request (synchronous)

- *rdp()*: non-blocking read of a tuple (asynchronous)
- *in()*: blocking read out and erase of a tuple, if the tuple is not present the process is suspended waiting for a tuple that matches the request (synchronous)
- *inp()*: non-blocking read out and erase of a tuple (asynchronous)

There are also four other primitives for bulk coordination to achieve significant gains in terms of efficiency for many coordination problems that involve managing more than one tuple in a single coordination operation [18]: instead of returning a single tuple, bulk coordination operations return an unified tuple list. In the event that is found no tuple, return with success an empty tuple list: then bulk primitive are always successful.

An element can be updated before deleting it and then reinserting the modified version. The *out(Tuple)* operation provides a key and the value of the arguments of the new tuple, while the other four operations specify the key and the arity (number of arguments) of the tuple to be read or consume. You can have duplicated tuples, so read or consumed tuples are not necessarily uniquely determined by provided key and arity. The operations *rdp(TupleTemplate)* and *inp(TupleTemplate)* guarantee to find a tuple that matchs with the request if this tuple has been inserted (but not consumed yet) in the space of tuples before the request was generated. In the Picture 1 you can see an example of a space of tuples in action [21].

1.2.3 Limitations of Tuple Spaces

One of the biggest advantages of the tuple spaces based interaction is that coordination is *information driven*: agents synchronize themselves; they also cooperate and enter into competition depending on the information available in a shared data space, so agents access it in associative mode by producing, reading or consuming information.

All this makes interaction protocols simple and expressive, but there is a problem: there is no way to separate how information is represented by how the informations are perceived by the agents (e.g. *Dining Philosophers Problem*, [3]).

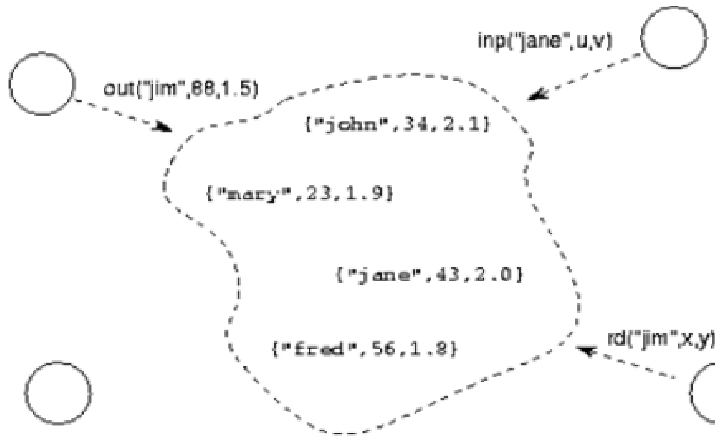


Figure 1.1: Example of use of a tuple space.

1.2.4 From Tuple Spaces to Tuple Centres

The limits described in the previous subsection can be overcome by maintaining separated information's representation from its perception in the agents communication space.

This result can be achieved by keeping the standard interface of tuple spaces and enriching it with the ability to define the behavior in terms of state transitions as a result of certain communication events.

This is the motivation that led to the *Tuple Centres* definition, which are tuple spaces whose behavior can be modeled in response to incoming or outcoming communication events, in accordance with the coordination requests specifications [12].

In this context, the Tuple Centres represent abstractions of general-purpose coordination, which can be understood as the interaction-oriented virtual machines to use as a base to build any kind of high-level abstraction for coordination between the components of a system.

1.2.5 Tuple Centre Definition

The **Tuple Centre** takes the concept of *tuple space*, specialize and extend it with a **specific behavior** which defines the behavior of the centers of tuples in response to incoming or outgoing communication events. The Tuple Centre's behavior specification is expressed through a specific Turing equivalent language [13] to define **reactions** (a series of activities to be performed) that can be associated with any communication event which can be present in the Tuple Centre. This allows to define any coordination law by modelling the agents's interaction space.

This language should allow:

- to define sets of operations (reactions) to run on a Tuple Centre
- to associate reactions to any incoming and outgoing communication events that may arise in the Tuple Centre

Each reaction can change the status of the center of tuples, such as inserting or consuming tuples, or access information related to the communication event that has triggered the reaction, such as the agent involved or the type of operation performed, etc. In this way you can model the Tuple Centre behavior as a simple primitives' result, in order to shape the Tuple Centre's semantics according to necessity.

Each event can trigger many reactions that will be performed before serving another communication event request by another agent; in this way, from the agent's point of view, the result of a communication primitive executed on a Tuple Centre is the sum of the primitives and of all the reactions that it triggers, and this result is perceived as a single state transition in the Tuple Centre. So, we can say that a center of tuple with an empty behaviour specification set is reduced to a simple space of tuples.

Thus, the Tuple Centres are a means for *data-driven* communicating, that keep all advantages resulting from it, and also provides some of the characteristics of *control driven* models, such as the complete observability of communication events, the ability to react in a selective manner to them, and implement coordination rules by manipulating the interaction space

[12]. While the basic model of the Tuple Centre is not tied to any specific language, to define the behavior in terms of reactions (or defining a set of *specification tuples*) TuCSoN adopts a specific language called **ReSpecT** [22].

1.3 Model, language, architecture and technology in TuCSoN

TuCSoN (*Tuple Centres Spread over the Network*) is a model for the coordination of distributed processes, as well as autonomous, intelligent and mobile agents [14].

The TuCSoN Tuple Centres adopt the **ReSpecT** language to define their behavior in terms of reactions (ie to define the set of *specification tuples*) [22].

1.3.1 TuCSoN Model

The TuCSoN 's base entities are:

- TuCSoN **Agents**: the *coordinables entities*
- **ReSpecT Tuple Centres**: the *means of coordination* (by default in TuCSoN model) [15]
- TuCSoN **Nodes**: represent the *topological abstraction* that host the Tuple Centres

All agents, nodes and Tuple Centres have an *universal identificator* in a TuCSoN system.

Being agents *proactive entities* while Tuple Centres are *reactive entities*, the coordinated entities require **coordination operations** in order to act on the means of coordination: these operations are given by the TuCSoN **coordination language**. The agents interact by exchanging tuples through Tuple Centres using the TuCSoN coordination primitives that all together define the coordination language. The Tuple Centres provide two distinct tuple spaces:

- the communication tuple based shared space (*ordinary tuples*)

- the space to programming the behaviour of the tuple based coordination (*specification tuples*)

The TuCSoN model has the following characteristics [23]:

- agents and Tuple Centres are distributed over the network
- Tuple Centre belong to nodes
- agents can reside anywhere on the network, and thay can interact with the centers of tuples hosted by any reachable TuCSoN node.
- agents can *move regardless of the device* where they are running [16]
- The Tuple Centres are associated in a permanent manner to a device, possibly mobile (the mobility of the Tuple Centres depends on the device on which they reside).

So, a TuCSoN system is a collection of TuCSoN nodes possibly distributed and they are associated with agents in execution on mobile devices that interact with Tuple Centres on the nodes.

1.3.2 Naming in TuCSoN

Each TuCSoN **node** in a TuCSoN **system** is uniquely identified by the pair

[*NetworkID*, *PortNumber*]

Where the parameters in the pair are described as follows:

- *NetworkID*: this is the IP address, or DNS address, of the device that hosts the TuCSoN node
- *PortNumber*: this is the port number where the TuCSoN *coordination service* listen the coordination operation invocations

So, the abstract syntax of a TuCSoN **node ID** hosted by a device connected to the netword *netid* on the port with number *portno* is:

[**networkid : portno**]

An **admissible name for a Tuple Centre** is any logical *ground*(that doesn't contains variables) term of the first order. Each node contains at most one center of tuples for each admissible name, then each Tuple Centre is uniquely identified by its name associated with the admissible identifier of the node.

So, the TuCSoN complete name of a Tuple Centre *tname* in a TuCSoN node [*networkid : portno*] is:

[**tname @ networkid : portno**]

This works as a global identifier for a Tuple Centre in a TuCSoN system.

An **admissible name for an agent** is any *ground*(that doesn't contains variables) term of the first logic order [17]. When an agent enters into a TuCSoN system, this system assigne an *universally unique identifier* (*Universally Unique Identifier - UUID* [24]) to that agent.

So, the complete name *agname* for an agent who has been assigned the UUID *uuid* turns out to be:

[**aname : uuid**]

1.3.3 TuCSoN Language

The TuCSoN **coordination language** allows agents to interact with Tuple Centres by performing *coordination operations.* TuCSoN provides to coordinated entities many *coordination primitives* that allows agents to write, read and consume tuples in tuple spaces, and to synchronize with them.

The coordination operations are built starting from the coordination primitives and *communication languages* [23] :

- the **tuples language**
- the **tuple templates language**

Both languages depend from the kind of Tuple Centre adopted by TuCSoN , in fact the default means of coordination in TuCSoN are **ReSpecT** Tuple Centres and the language of the tuples and tuple templates are both logic based. More precisely:

- each Prolog Atom can be an **admissible TuCSoN tuple**

- each Prolog Atom can be an **admissible TuCSoN tuple template**

As a result, TuCSoN default languages about tuples and tuple templates concide.

A TuCSoN *coordination operation* is invoked by a **source agent** on a determined **destination Tuple Centre**, which is in charge of its execution.

Each TuCSoN operation has two phaases:

- **invocation**: the source agent request to the destination Tuple Centre, the request transport all the informations about the invocation
- **completion**: the reply from the Tuple Centre destination to the source agent, the response includes all the informations about the execution

The abstract syntax of a coordination operation *op* invoked on the destination Tuple Centre who has the complete name *tcid* is:

[**tcid ? op**]

Given the structure of a complete name for a Tuple Centre, the *general abstract syntax* for a coordination operation TuCSoN will be:

[**tname @ netid : portno ? op**]

Finally, the TuCSoN coordination language provides all the *coordination primitives* (eg: *basic coordination primitives* [20], *bulk coordination primitives* [18], *uniform coordination primitives* [23]) needed to build coordination operations.

1.3.4 TuCSoN Architecture

A TuCSoN system is characterized by a collection of TuCSoN nodes (possibly distributed) on which it is running a TuCSoN service.

A TuCSoN node is characterized by a device connected to the network on which it is running the service, and a network port where the TuCSoN service is listening for incoming requests. In principle, many TuCSoN nodes can running on the same device connected to the network, each of which is listening on a different port [23].

The TuCSoN default port number is *20504*. Thus, as seen above, an agent can invoke operations by issuing the following request:

[**tname @ netid ? op**]

without specifying the port number *portno*, it means that the agent wishes to invoke the operation *op* on the Tuple Centre *tname* of the default node *netid* hosted by the device connected to the network *netid* on port *20504*.

In principle, any other port can be used for a TuCSoN node. The fact that a TuCSoN node is available on a device connected to the network does not imply that a node is also available in the same unit on the default port (in practice, it is no certain that there is the default node).

Given an acceptable name for a Tuple Centre *tname*, the Tuple Centre *tname* is an **admissible Tuple Centre**. The *coordination space* of a TuCSoN node is defined as a collection that contains *all admissible Tuple Centres.* Each TuCSoN node provides to agents a complete coordination space, so any coordination operation can be invoked on any admissible Tuple Centre belonging to any TuCSoN node.

Each TuCSoN node defines a **default Tuple Centre**, that reply to all the invocations received by any node of any operation that does not specify the destination Tuple Centre: the *default Tuple Centre* for each TuCSoN node is called *default.* As result, agents can invoke operation by the following request form:

[**@ netid : portno ? op**]

without specifying the Tuple Centre *tname*, it means wishing to make the operation *op* on the *default* Tuple Centre node *netids* hosted by the device connected to the network *netid* on the port *portno.*

Combining the concepts of default Tuple Centre and default port, agents can also invoke operations of the following request form:

[**@ netid ? op**]

with the meaning intending to invoke the operation *op* on the *default* Tuple Centre node *netid* hosted by the device connected to the network *netid* on the port *20504*.

1.3.5 TuCSoN Coordination Space

The TuCSoN global coordination space is defined at any time from the collection of all available Tuple Centres on the network, and hosted by an identified node by their full name [23]. A TuCSoN agent running on any device connected to the network has at each instant the full TuCSoN global coordination space available for its coordination operations through invocations by the following form:

[tname @ netid : portno ? op]

that invoke the operation *op* on the Tuple Centre *tname* provided by the node *netid* on the port *portno*.

Given a device connected to the network *netid* that hosts one or more TuCSoN nodes, the TuCSoN **local coordination space** is defined at any time from the collection of all Tuple Centre made available by all TuCSoN nodes hosted by *netid* By exploiting the concepts of default node and default Tuple Centre, any agent can exploit the local coordination space for all request forms previously seen or any admissible request invocation [23].

1.3.6 TuCSoN Technology

TuCSoN is a **Java and Prolog based middleware**. TuCSoN relies on Java *tuProlog* for:

- first order logic tuples
- analyze and identify primitives
- ReSpecT specification language and virtual machine

The TuCSoN middleware provides:

- **Java APIs** in order to exend Java programs with TuCSoN primitives
- **Java Classes** in order to program TuCSoN agents with Java language
- **Prolog Libreries** in order to exends *tuProlog* programs with TuCSoN coordination primitives

Given any network-connected device that is running on a Java virtual machine (JVM), on it you can start a TuCSoN node to provide a TuCSoN **service**. The service on the node must:

- listen to the invocations of operations on the incoming port associated with the device TuCSoN service

- send received invocations to the destination Tuple Centres
- return operations' completions

A TuCSoN service running on a node provides the full coordination space.

The centers of tuples in a node can be in every moment *actual* or *potential*:

- **actual Tuple Centres**: eligible Tuple Centres that already have a reification as an abstraction at runtime
- **potential Tuple Centres**: eligible Tuple Centres that have not yet as a reification abstraction at run-time

The service on the node have to change potentials Tuple Centres to actuals as soon as the first operation on them is received and served.

Details for using the TuCSoN infrastructure, both to initialize the service agents to handle Java and Prolog or to take informations about tools provided by TuCSoN package, can be found in the guide [23].

1.4 Background Knowledge for Cassandra

In this section will be described all background knowledge in order to understand Cassandra .

Cassandra will be used in the module's model in order to give a technological solution to data storage issues that will be studied in following chapter.

1.4.1 CAP Theorem

A distributed system cannot simultaneously provide more than two out of the following three guarantees:

- **Consistency**
- **Availability**
- **Partition Tolerance**

In the following picture will be described the previous property about distributed systems relating to following four important data storage distributed system:

- *Big Table - MongoDB Redis*: management system of non-relational databases, document-oriented, type NoSQL [25]
- *RDBMS*: database management system (DBMS) that is based on the relational model, the full description is **Relational DataBase Management System** [28]
- Cassandra : non-relational database management system optimized for handling large amounts of data; it will be described in following sections [27]
- *Dynamo*: fast, fully managed NoSQL database service that makes it simple and cost-effective to store and retrieve any amount of data, and serve any level of request traffic [26]

So, starting from these described distributed system, the graph that explain better the main property is:

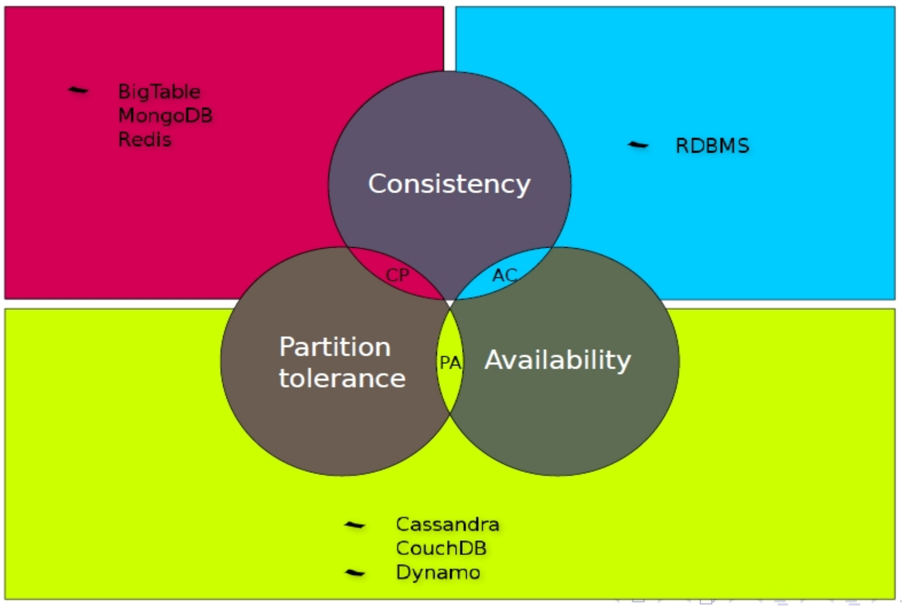


Figure 1.2: Table Data about Distributed Systems Properties.

As you can see in the above picture, there are three possibility about the distributed system properties pair:

- **CP**: pair that explain that the distributed system has the **consistency** and **partition tolerance** properties
- **AC**: pair that explain that the distributed system has the **availability** and **consistency** properties
- **PA**: pair that explain that the distributed system has the **partition tolerance** and **availability** properties

To understand and interpret this theorem you have to undestand what are *partitions*. In a distributed system *partitions* are *unavoidable*.
Then you have to think to **CAP** as the following logic equation:

$$\mathbf{P} = (\ \mathbf{A\ XOR\ C})$$

So, as you can see in the previous logic equation, as a *partition* occurs, the tradeoff is between:

- **A - availability**: in terms of *cost of inconsistency*
- **C - consistency**: in terms of *cost of downtime*

1.4.2 Variation over CAP Theorem

A better classification for real-life systems have been introduced with a variation over the **CAP Theorem** seen in the previous section.

This variation is based on the following condition: *if there's no partition, the tradeoff is about how much Latency the system employs to achieve a certain degree of Consistency.*

Then, there is a logic equation better that the previous one and it is:

P ? ((A OR C) : (L OR C))

So, as you can see in this logic equation, there are two cases, dependently from the presence of a *partition*, in the case of ***one partition**, the tradeoff is between **Availability (A)** and **Consistency (C)***, while *if there is no partitions (so in **otherwise cases**), the tradeoff is between **Latency (L)** and **Consistency (C)**.*

Starting from this variation about the seen theorem, there is a further system classification, in fact you have the following policy:

*Systems following **P ? C** depend on **:C***

So there is the following classification:

- **[P ? C : C]** : reliable systems (eg: DBMS) so this classification relating to *fully ACID systems* and traditional *RDBMS*
- **[P ? A : C]** : systems that *give up on **Consistency** as partitions occur*
- **[P ? A : L]** : *high availability systems focused on **low latency** and **high scalability***

The **ACID** property is related to a group of four properties that an operation can have and the meaning is:

- **A - atomic**: operation that doesn't break indivisible consistency ("all or nothing")
- **C - consistent**: committed updates don't break domain constraints
- **I - isolated**: doesn't break serializable consistency
- **D - durable**: committed updates are persistent

These properties are the same about the transaction operation in the *DataBase space* because this section is related to systems that use database and similar structures in order to storage data. There are also three property that a system can have, relating to all properties previously seen; these properties are:

- **Basic Availability**: availability is the main focus of the system
- **Soft State**: state of the system can change anytime, even without inputs
- **Eventually Consistent** : system will become consistent after a long enough timeframe without inputs

These properties are derived by all previously seen properties and conditions and these are very important in order to obtain a better data storage distributed system as we want to have in the module studied in the following chapter.

1.4.3 Limits of RDBMS

The main limits of **RDBMS** are related to the **Two Phase Commit** algorithm that put in evidence all limits and problems derived by most system characteristics, like for example the huge quantity of data to storage.

In particular, will be explained all properties about the **Two Phase Commit** and will be described all problem and all its limits because this is the most used algorithm in **RDBMS**.

The **Two Phase Commit** is an algorithm taking place in a *Distributed Atomic Transaction* (*DAT*). This algorithm has the aim to coordinate all processes that participates in a *DAT*. This algortithm is a type of *Consensus*

Protocol in order to achieve the described aim and it tolerates a wide fraction of failure configurations about participant processes.

The **Two Phase Commit** is composed by two phases:

- Phase 1 - **Commit Request Phase** : *the Coordinator sends a query to commit to each Participant of the transaction and waits for all replies*; *each Participant performs the transaction up to the point in wich it's asked to commit and send a reply*, this reply can be a message of:

  - **Agreement** : which means that the *Participant would commit the transaction*
  - **Abort** : which means that the *Participant would roolback the transaction*

- Phase 2 - **Commit / Rollback phase** : *the Coordinator decides wheter to commit or rollback the transaction, and issues the outcome to each Participant*; *each Participant follows the issued commit or rollback, releasing lock and resources*; in the end, *each Participant sends an acknowledgement back to the Coordinator.* The *Coordinator completes the transaction when all acknowledgements have been received*

From this description it can be seen the **Consistency** costs a lot because

- *it block* the execution in many parts of each phase
- the *Coordinator is a* ***Single Point of Failure***
- *Horizontal Scaling is not so effective*

All solutions in the **RDBMS** world are related to optimizations about the *recursive approach* (2PC optimizations like Recursive 2PC, etc.) or a variation of a known algorithm (for example the *Three Phase Commit* that's more resilient to failures but still blocking).

So, it's evident that in some scenarios the **ACID** is too much for this kind of **DBMS** and it needs to introduce a new architectural model that have to be more efficient and more powerfull in actual scenarios.

1.5 Cassandra

In this section will be described Cassandra , a non-relational database management system optimized for handling large amounts of data that can work in a distributed management system maintaining all its features [27].

A particular feature about this DBMS is that **Cassandra 's Data Model** is based on *Google's **Bigtable*** and its **Distribution Model** is based on *Amazon's **Dynamo***. This is a very important feature because from these model derive all the following key properties about Cassandra :

- **NoSQL Data Model** : deriving from *Google's **Bigtable*** this model doesn't adopt SQL Data Model so its data model can be seen as a *multidimensional hash-map* that have to contain all system's data
- **Decentralized Model** : deriving from *Amazon's **Dynamo*** this DBMS Model is optimized as a distributed system where *nodes are peers, each node can satisfy any request and there is no single-point of failure*
- **Elastic Scalability** : from two "parents system" derive also this property that give to the system the *horizontally scalability, both up and down, almost linearly*
- **Fault Tolerance** : thanks to the mix of those two system model, this property is giveng by *data replication* that allows *to add and replace any node without downtime*
- **High Availability** : given from the *reliability* of this model and its elastic runtime adapting to any distributed configuration given by deriving from *Amazon's **Dynamo***
- **Tuneable Consistency** : in order to obtain the best kind of consistency, this property ca be given by two diffent ways:
  - by **cluster configuration**: working on many distribution factors (eg replication factor)
  - by **client, per-operation**: working on many model levels basing the choice on many requirements and issues (eg consistency level)

1.5.1 Data Model

As previously seen, the **Cassandra 's Data Model** derive from *Google's **Bigtable*** so it has almost all features about that model [29].

The most important entities in the **Cassandra 's Data Model** are:

- **Keyspace**: outermost container for data, it is similar to the table schema in *Relational DataBase* (*RDB*) world
- **Column Family**: collection of columns, it is similar to table in *Relational DataBase* (*RDB*) world
- **Column**: map from row keys to values; values updates are timestamped and all timestamps are provided by clients that communicate with the system
- **SuperColumn**: special kind of columns wich hold normal columns; this special columns can have also special functions about data manipolation

There are also many characters to underline because these characters make this model better thant many others; these characters are related to the schema, to amount of data and to the sorting about data.

In the firse case you have that different rows can have different columns in the same *Column Family*, this character allow to define this model's schema as a *schema-less "tables"*, so it give to this schema many advantages relating to constrains about columns and their families. In the second case you have that the data growth is often in the "column direction" while in the *RDB* world the data growth is related in the "rows direction". The last key character about this schema is related to the sorting of data, in fact, in this schema columns are *sorted by their name* that make more efficient all column-slice queries and it is heavily exploited in data design.

In the end, this *data model* is classified as **JSON-ified** because all entities and features are represented and decoded by using the **JSON language** because this language has many important features and advantages in network communication in order to obtain a better efficency and a better reliability in the distributed data storage system.

1.5.2 Data Distribution Model

As previously seen, the **Cassandra 's Distribution Model** derive from *Amazon's **Dynamo*** so it has almost all features about that model [26]. This model's **Data Space** is represented as a ring of tokens (this number depends on the distributed system) where *each token cooresponds to a row key*. This **Data Space** is sliced into ranges, where the number of ranges is equal to the number of nodes, so there is a binary correspondance between nodes and ranges. Thanks to that feature, each *node covers a slice of the **token space*** and the *mapping operation between row keys and tokens* is the aim of a component called **Partitioner**; consequently this mapping affects data distribution among the nodes.

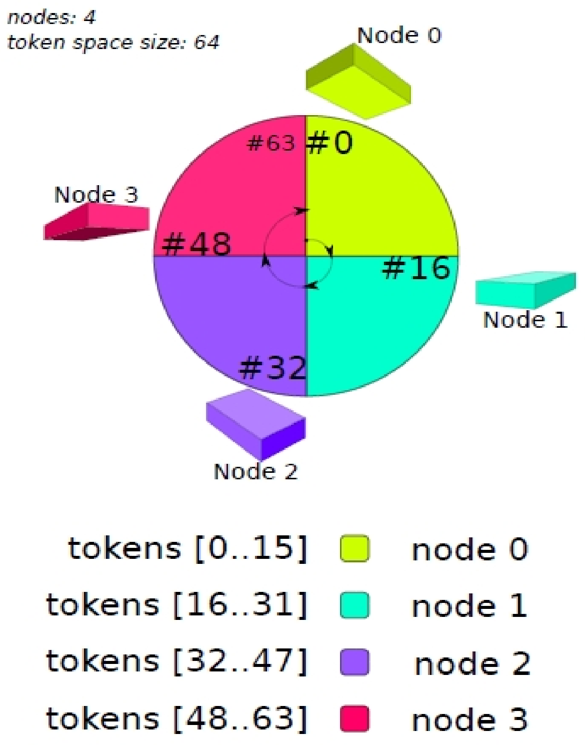


Figure 1.3: Example of Cassandra's Data Distribution Model.

This distribution model is based on the **Row Oriented Sharding and Replication of Data** that allows many important functionalities in order to manipolate the **Data Space**.

In the end there are the following main knobs relating to the possible configurations:

- **cluster-wide configuration**: in this configuration the **Partitioner** *controls how data is partitioned* (for example: which node belongs k-row key?)
- **per-keyspace configuration**: in this configuration the **Replication Factor** *controls how many replicas are stored* and the **Replication Strategy** *controls in which nodes replicated data goes*

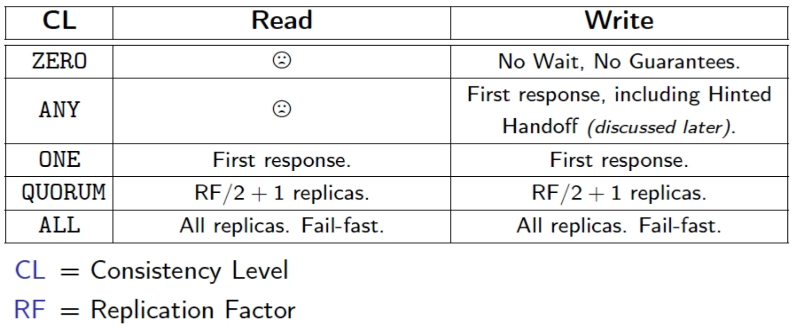
1.5.3 Client Requests

First of all, if a *client* has to send a request to a *node*, it needs to connect to the target *node* and after that operation, the *client* can perform the request.

The *node* plays the *Coordinator* role for this operation because the message exchanging between these two entities will be managed by the entity that have to sadisfy the request (the *node*).

Then, the *node* acts to fullfill the received request and notify the result to the sender *client* if the request have been executed correctly or not. In all these phases, the communication to other nodes is generally involved.

Each client request has an associated Consistency Level: the *minimum number of replicas that should respond.* These levels are described for read and write operations in the following table.



CL	Read	Write
ZERO	☹	No Wait, No Guarantees.
ANY	☹	First response, including Hinted Handoff *(discussed later)*.
ONE	First response.	First response.
QUORUM	RF/2 + 1 replicas.	RF/2 + 1 replicas.
ALL	All replicas. Fail-fast.	All replicas. Fail-fast.

CL = Consistency Level
RF = Replication Factor

Figure 1.4: Clients - Nodes Operations' Consistency Levels.

From the above picture you can see that the **write operation** has following properties about its executions and the consistency level:

- the consistency level to *ALL* or *QUORUM* guarantee writes to a sufficient number of replicas
- *QUORUM* statistically has a lower latency then others and is more fault tolerant then *ALL*
- failed writes don't cause roll-backs in any of the eventual nodes where the write succeeded
- Cassandra guarantees *atomic writes on a single node, row key basis*

There is a particular situation when the *Coordinator detects that a replica node for the write operation is down* and this write operation *is set to the ANY or higher consistency level.* In this situation the *Coordinator writes down an **hint** for that operation* that will be ***handoff** to the target node once it will become available again.*

This operation is called **Hinted Handoff** and is executed because it *reduce the time required for a temporarily failed node to become consistent again* and *provides **extreme write Availability** when Consistency is not required (Consistency Level ANY).*

Like any other operation, this one has some important issues to consider:

- When a failed node becomes available again, could be flooded by hints from other nodes?
- It's possible to disable **Hinted Handoff** or reduce its priority of over "normal" writes?

In order to obtain an efficient **Hinted Handoff** operation you have to consider these issues basing on requirements and use case.

Like write operations there are many issues related to **read operations** that can be done. In this phase the *Coordinator contacts only a number of replicas specified by the Consistency Level* basing on the number of replicas involved in that process, in fact:

- **only one replica involed**: a ***background thread*** *performs* ***Read Repair*** *operation to ensure Consistency with other replicas*
- **more then one replicas involed**: if the *involved replicas are inconsistent, a foreground (blocking)* ***Read Repair*** *is performed*, *a* ***background thread*** *performs* ***Read Repair*** *between all replicas* and is return the *most recent value*

The **Read Repair** is a *timestamp-based* way to achieve **replica conciliation**, as seen above, these timestamp are provided by *clients* and, because they are part of the system, they *must have* ***low clock drifts*** (for example this issue is solved by using **vector clocks** like in Dynamo [26]).

In the end, we have to answer to an important question:

Where to put ***latency?***

For this issue Cassandra choose *low latency writes over low latency reads* (**Read Repair over Write Repair**). The adopted model is the **Cache write-back model** (Borrowed from *BigTable* [25]) where:

- writes are written to a *commit log*
- writes are written to in-memory tables (*memtables*)
- writes are flushed in persistent storage to read-only tables (*SSTables: Sorted Strings Tables*) that are periodically optimized.

So, the **Consistency** can be:

- **Weak**: when you have that *(Write + Read) is lesser then* ***Replication Factor***, in this case *clients may read stale values*
- **Strong**: when you have that *(Write + Read) is greather then* ***Replication Factor***, in this case *clients will always read the most recent write*

In the following pictures is showed an example of **Strong Consistency** and the impact of the **Consistency Level Choice** on the system.

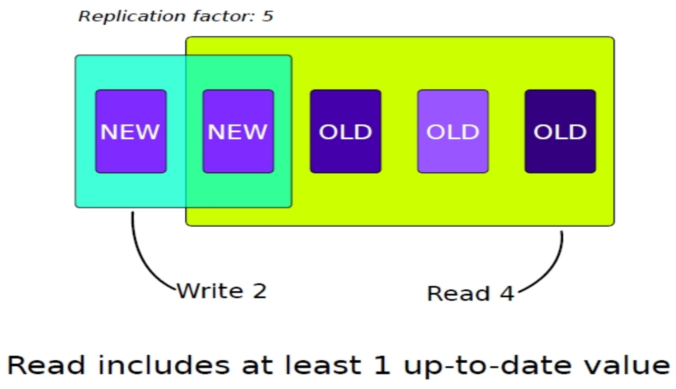


Figure 1.5: Strong Consistency Example.

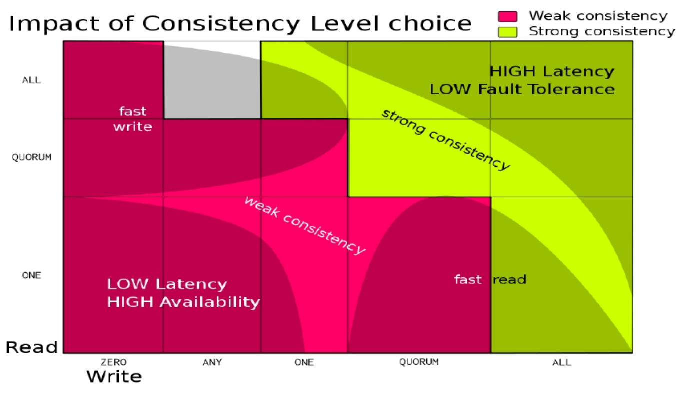


Figure 1.6: Impact of the Consistency Level Choice.

All details about model and implementation of Cassandra can be found in its website [27] like the other system seen in this chapter (*Google's **Bigtable*** [?] and *Amazon's **Dynamo*** [26]).

Chapter 2

JADE

2.1 What is JADE

JADE (**Java Agent DEvelopment Framework**) [30] is a Java-based framework to develop agent-based applications in compliance with the **FIPA *specifications*** [31] for *interoperable*, *intelligent*, *multi-agent systems.*

As an **agent-oriented middleware**, JADE pursues the twofold *goal* of being:

- a **full-fledged FIPA-compliant agent platform**. Hence, it takes in charge all *application-independent aspects* (for example agents life-cycle management, communications, distribution transparency, etc.) needed to implement a *MAS* [32]
- a simple comprehensive **agent development framework**. Therefore it provides *Java developers* a set of *APIs* (it stands for *Application Programming Interface*, so all interfaces needed to developers) to build their own customizations

Being *fully implemented in Java*, JADE is a notable example of a **distributed**, **object-based** and **agent-oriented infrastructure**, hence an interesting example about *how to face a design/programming paradigm shift.*

Being compliant to **FIPA** standards, JADE is a **complete** and **coherent agent platform** providing all the necessary facility to *deploy* **MASs**.

As follows all *main features* about JADE are itemized, in fact, JADE offers:

- a **distributed agent platform**, where *distributed* means that a *single JADE system can be split among different networked hosts*
- **transparent**, **distributed message passing interface & service**
- **transparent**, **distributed naming service**
- **white pages & yellow pages** discovering facilities
- **intra-platform agent mobility** (contex, to some extent)
- **debugging & monitoring** facilities

All other details about JADE and **FIPA** can be found at the specific websites: JADE **Official Website** [30] and the **FIPA Official Website** [31]

2.2 JADE Architecture

The following picture represents the **main JADE architectural elements**.

An application based on JADE is made of a set of components (the **Agents**) each one having a *unique name*. **Agents** *execute tasks* and *interact by exchanging messages.*

Agents live on top of a **Platform** that *provides them with basic services* such as message delivery. A **Platform** is composed of one or more **Containers** each one *can be executed on different* ***hosts*** thus achieving a **distributed platform** and each **Container** can *contain zero or more* **Agents**.

For instance, with reference to the picture below, **Container** *Container 1* in host *Host 3* contains **Agents** *A2* and *A3*. Even if in some particular scenarios this is not always the case, you can think of a **Container** as a *JVM* (it stands for *Java Virtual Machine*, the virtual machine that execute the *Java Platform*) so there is the relation:

One JVM - One Container - Zero or Many Agents

A special *Container* is the *Main Container* existing in any platform. The *Main Container* is itself a *container* and can therefore *contain agents*, but differs from other containers as:

- It must be the *first container to start in the platform* and *all other containers register to it at bootstrap time*
- It includes two **Special Agents**:
  - the **AMS** (it stands for *Agent Management System*) that represents the *authority in the platform* and is the *only agent able to perform platform management actions* such as starting and killing agents or shutting down the whole platform (normal agents can request such actions to the **AMS**)
  - the **DF** (it stands for *Directory Facilitator*) that *provides* the **Yellow Pages Service** where agents can publish the services they provide and find other agents providing the services they need

It should be noticed that if another **Main Container** is started, as in host **Host 4** in the following picture, this constitutes a *new platform.*

All details about this architecture will be described in following sections.

2.2.1 JADE & FIPA

According to **FIPA**, the **Agent Platform** can be splitted on several hosts given that:

- each host acts a **container of agents**, that is, provides a complete *runtime environment* for JADE **agents execution** (for example the lifecycle management, message passing facilities, etc.)
- (at least) one of these containers is the **main container** (actually, the *first started*), responsible to *maintain a registry of all containers* in the same JADE platform (through *wich agents can discover each other*)
- hence, it promotes a **P2P** (it stands for **Peer to Peer**, the complete description can be found at the specific Wikipedia web page [33]) **interpretation** of a **MAS**

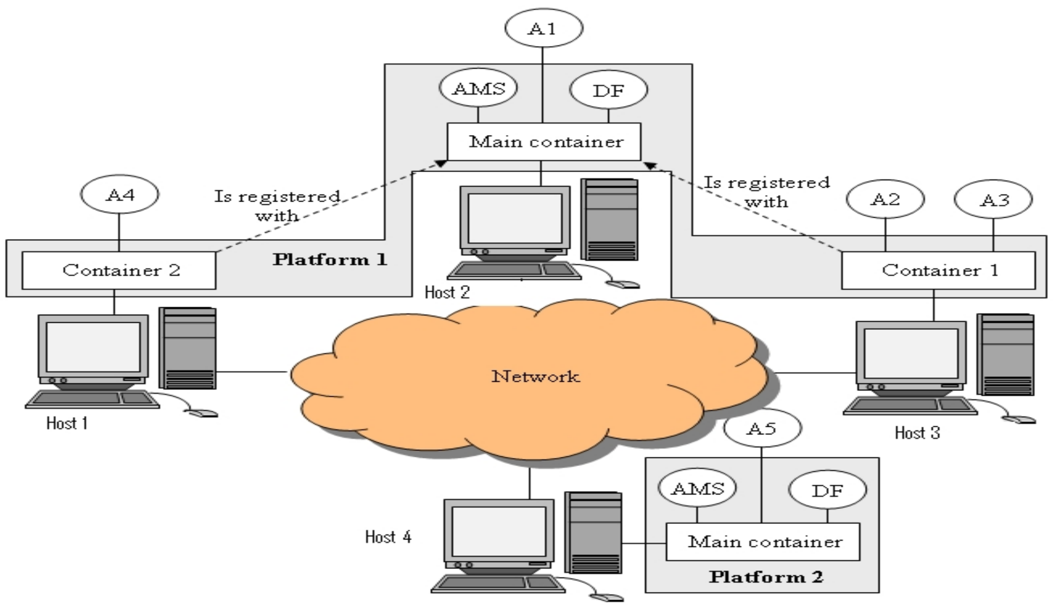


Figure 2.1: JADE Architectural Model

Then, after the **Agent Platform** description, we introduce the JADE **AMS**, that is another particular part of a JADE system. So, for a given JADE platform, a single **AMS** exists, wich:

- *keeps track of all agents in the same JADE platform*, even those *living in remote containers*
- should be contacted by JADE **agents** prior to any other action (they *do not exist* until *registered by an* **AMS**)
- hence, provides the **White Pages Service**, that is a ***location-transparent naming service***

Another JADE foundamental components is the **DF** (it stands for **Directory Facilitator**); a singleton **DF** exists for each JADE platform, that:

- *keeps track of all advertised services provided by all agents in the same JADE platform*

- should be contacted by JADE **agents** who wish to *publish their capabilities*
- hence, provides the default **Yellow Pages Service**, according to the ***publish & subscribe paradigm***

The last JADE foundamental component is the **ACC** (it stands for **Agent Communication Channel**) that is the agent that *provides the path for basic contact between agents inside and outside the platform*; so it is the *default communication method which offers a reliable, ordered and accurate message routine service* and it must also *support all functions for interoperability between different agent platforms.*

So, for a given JADE platform, a **distributed message passing system** exists, which is the **ACC**:

- it *controls all exchange of messages within the JADE platform*, be them local or remote
- it implements all the needed facilities to provide *asynchronicity of communications*
- it manages all aspects regarding **FIPA ACL** (it stands for **FIPA Agent Communication Language**) **message format**, such as serialization and deserialization

In the end, the JADE **FIPA Architecture** is showed in the following picture, that contains the normative and optionals services given by the **FIPA required services**.

2.2.2 JADE Agents

Being JADE an **object-based middleware**, JADE **agents** are first of all *Java objects*:

- user-defined agents must extend the specific *Java class*, thus inheriting some ready-to-use method
- a JADE **agent** is *executed by a single* ***Java thread*** (there is an exception, though)

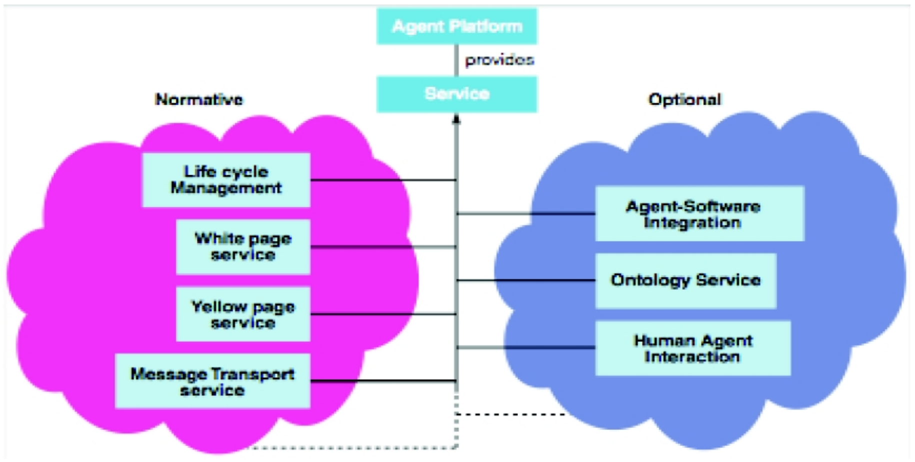


Figure 2.2: FIPA required services

JADE **agents** have a wide range of features enabling their **autonomy**, despite being still *Java objects*, so:

- all JADE agents must have a **globally unique name** (defined as **aid**), which is, by default, the concatenation by simbol **@** of their **local name** and the JADE **platform name**
- **agents bussiness logic** must be expressed in terms of **behaviours**
- JADE **agents** can communicate by exchanging **FIPA ACL messages**

Thus, according to **FIPA**, a JADE **agent** can be in one of several states during its lifetime:

- **Initiated**: the agent object has been **built**, but *cannot do anything since it is not registered to the **AMS** yet, so it has no **aid** even*
- **Active**: the agent is **registered** to the **AMS** and *can access all JADE features*, in particular, it is executing its behaviours
- **Waiting**: the agent is **blocked**, *waiting for something to happen and to react to*, tipically this is an **ACL message**

- **Suspended**: the agent is **stopped**, therefore *none of its behaviours are being executed*
- **Transit**: the agent **has started a migration process** and *it will stay in this state until migration ends*
- **Unknown**: the agent is **dead**, so it *has been deregistered* to the **AMS**

Thus, the lifecycle of a JADE **agent** can be described as the following picture, where are described all states and transitions of an agent's state, according to the **FIPA**.

So, as follows, is showed the **FIPA Agent Platform Life Cycle**:

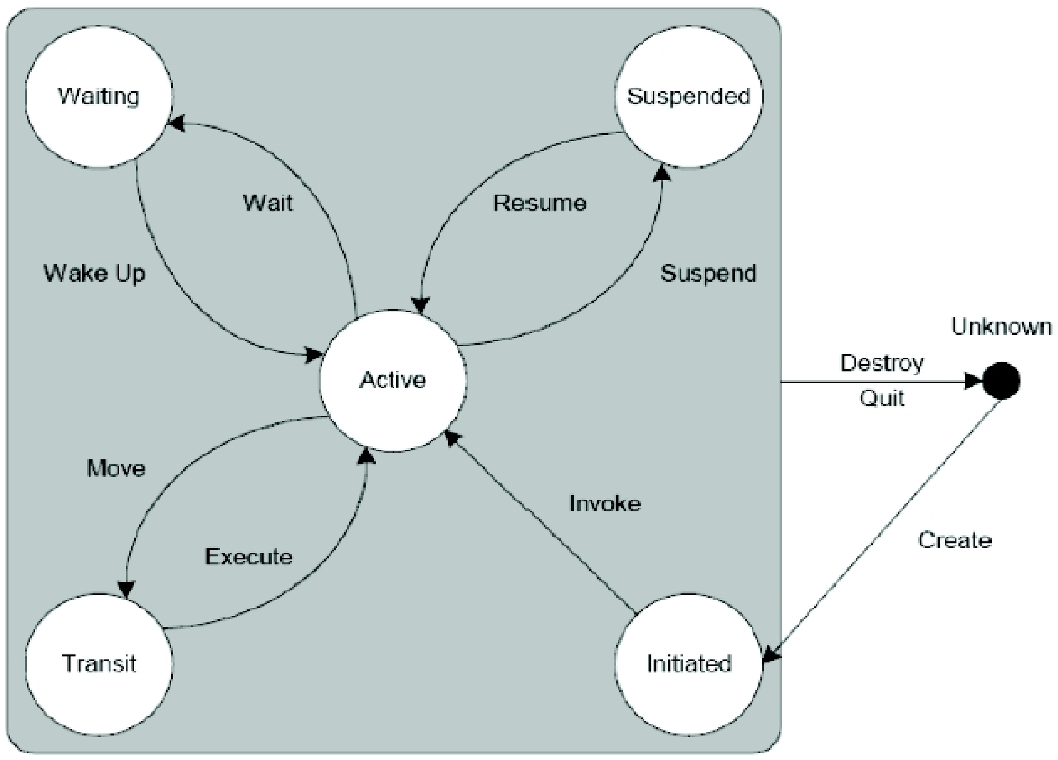


Figure 2.3: FIPA Agent Platform Life Cycle

After the agent lifecycle's description, we will treat the description about the **agent behaviours description** in order to explain all details about these entities. By definition, JADE agents are **autonomous entities**, therefore they should act *independently* and in *parallel* with each other, so the need for *efficency* drives toward the execution of JADE agents as a single **Java thread** each but, however, agents need to **perform complex activities**, possibly composed by **multiple tasks**, even **concurrently**.

Thus, theis contrasting needs to **behaviours** in order to *conciliate* them. A behaviour can be seen as **an activity to perform with the goal of completing a task**. It can be represented both as a **proactive activity** *started by the agent on its own*, as well as a **reactive activity** *performed in response to some event* (for example in response to a timeout event, messages event or any other event relating to agents).

However, JADE **implements behaviours as Java objects**, which are executed concurrently (still by a *single Java thread*) using a **non-preemptive**, **round-robin scheduler** that is internal to the agent class but hidden to the developer.

Thus, the JADE **multi-tasking, non-preemptive scheduling policy** is described as showed in the following picture:

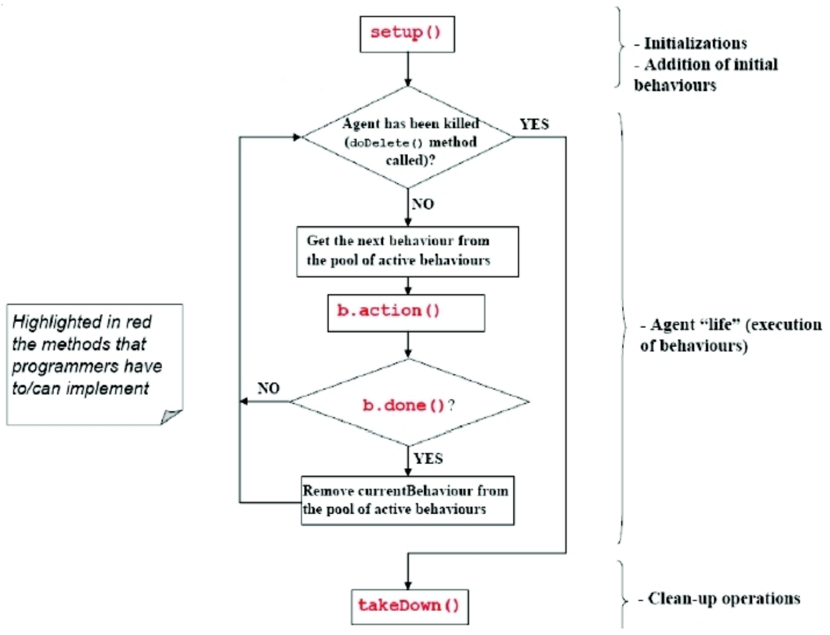

setup()
- Initializations
- Addition of initial behaviours
Agent has been killed (doDelete() method called)?
YES
NO
Get the next behaviour from the pool of active behaviours
b.action()
Highlighted in red the methods that programmers have to/can implement
- Agent "life" (execution of behaviours)
NO
b.done()?
YES
Remove currentBehaviour from the pool of active behaviours
takeDown()
- Clean-up operations

Figure 2.4: JADE multi-tasking, non-preemptive scheduling policy

2.2.3 FIPA ACC

According to the **FIPA** specification, JADE agents communicate via **asynchronous message passing** where:

- each agent has a *message queue* (a sort of *mailbox*) where the JADE **ACC** delivers **ACL** *messages* sent by other agents
- whenever a *new entry is added to the mailbox*, the receiving agent is **notified** so, *it does not need to block nor* to poll either

These specifications about the **message passing** are valid and must be respected *when* the agent actually process a message is up to the agent itself (or the programmer), for the sake of agents **autonomy**

Thus, the **ACL compliant messages** have a crucial role in order to obtain the situation where each agent is able to understand each other. So, they need to have a defined *format* and *semantic* relating to messages.

Hence, an **ACL message** contains the following foundamental informations:

- **sender**: the agent *who sends the message*; this field is automatically set by the middleware component where the sender agent live
- **receiver**: the agent *who the message targets*; this field is setted by the sender agent because it may be many different target agents
- **performative**: the *name of the communication act the agents want to carry out*; this field is constrained by a **FIPA** ontology
- **content**: the *actual information conveyed* by the message
- **language**: the *syntax* used to encode to the **content**
- **ontology**: the *semantics* upon which the **content** relies

Logically, there are many other fields but they depend on the technology progress of JADE so, the itemized fields are the foundamental fields that must be present in all JADE versions.

After the previous description about the **FIPA communication model**, the following picture shows a model abstraction about it in order to represent all itemized foundamental fields about a message and its communication.

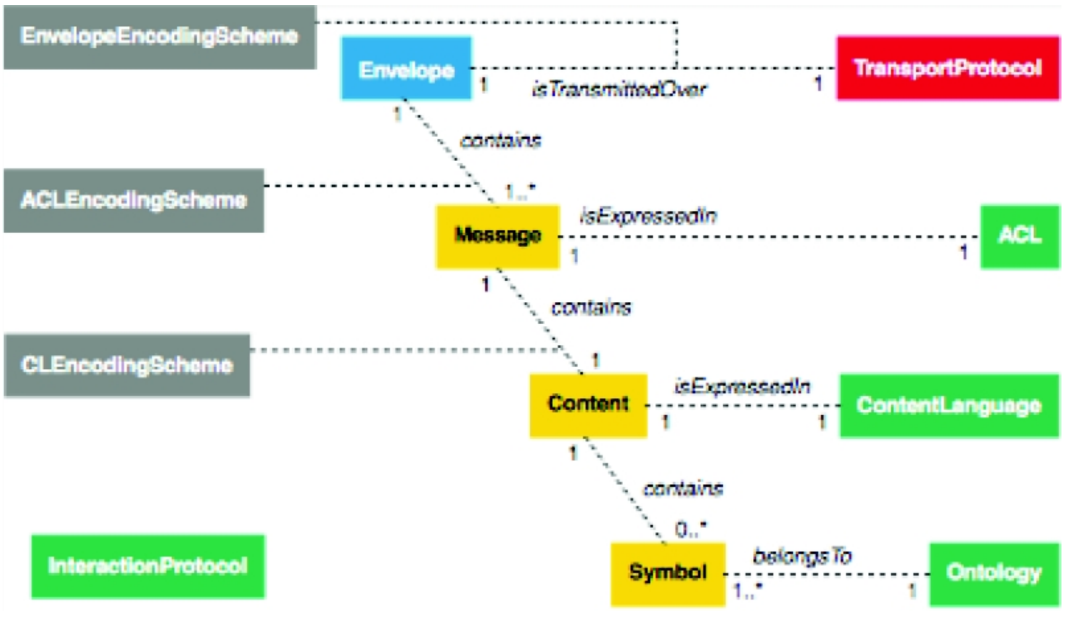


Figure 2.5: FIPA communication model abstraction

In the end, JADE **agents** have a couple of methods in order to guarantee a stable interaction mode; these foundamental communication primitives are:

- **send**: to *send a message* to a implicity specified recipient agent
- **receive**: to *asynchronously retrieve* the first message in the mailbox, logically, if the mailbox contains only one message, this message will be retrieved to the target agent involved in the communication
- **timed receive**: to perform a *timed, synchronously receive* on the mailbox in order to obtain the situation where the *timeout causes agent to wake up*, so it has to retrieve the messages contained in the mailbox

- **selective receive**: to *retrieve a message* from the mailbox which *matches a given message template*, so the message queue order is bypassed by the target agent involved in the communication because it selects the message to retrieve

All these methods are **distribution-transparent**, that is they choose the proper address and transport mechanism based upon sender and receiver locations in order to obtain the best communication situation.

Chapter 3

Naming, Locator & Research Service Structure & Interaction Model

In this chapter it will be introduced the general model for providing services related to **naming**, **location** and **research service**.

These services are part of a separate module that can be used with any middleware, less than small changes relative to the structural details of the application context of the middleware considered. In the case treated, this module is used with the middleware TuCSoN , introduced and described in the first chapter.

This model has been studied starting from the previously seen JADE system because that system is an efficient distributed middleware for distributed system. This system is JADE like in terms of *layer vertical modeling* because it is a good solution relating to system network distribution.

Comparing with the JADE system, this model's characteristics have been increased in order to obtain a better model, in particular it has been studied in order to increase the characteristics about the network distribution because, as described in the previous chapter, JADE has many constrains that limit the possible network distribution of that system. In fact, this model is more distributed-oriented than JADE because all lower layers can be distributed excep the upper layer because, as you can see in the following sections, it must be located on the consumer because it is the layer that has the aim to manage the communication with the consumer.

Another advantage of this model compared to jade is the complete independence of the layers by their location in order to remove all the constraints that made jade less network distribution-oriented of internal parts of the middleware, therefore this model has a higher efficiency in terms of computational load.

Also there is another foundament characteristic of this model: the independence of this module by the consumer middleware.

The decision to design this module as a separate module from the context of specific use (varying the model that will be introduced for TuCSoN the same logic can be adapted to any type of middleware) was taken to obtain the maximum *orthogonality* between the *consumer middleware* (TuCSoN in the considered case), and the *service module* introduced in this chapter. The orthogonality is guaranteed by the characteristic malleability of the structure of the considered module because this module template is suitable for multiple contexts of use.

As you will see below, this module can be contained entirely on the consumer, or may be distributed, in this case there will be only a part of the module of a consumer (probably would be on the consumer only the part of the module that it intends to use, or the module's most used part) and the rest of the module would be placed in another point of the system and on demand from the consumer involved in the distributed architecture of that specific instance of the module.

3.1 Main Model

The following picture shows the architectural main model of the considered module. This model has been studied in order to have a common external interface to customers and three specialized modules inside:

- a module related to the **Universal Naming Service**: service that provides all functions about the *naming system* (for example: giving an universal identificator to a customer)

- a module related to the **Universal Locator Service**: service that provides all functions about the *locator system* (for example: registering the current customer's position)

- a module related to the **Research Service**: service that provides all functions about the *research system* (for example: research the customer's position starting from its universal identificator)

All these modules will be described in details in following sections.

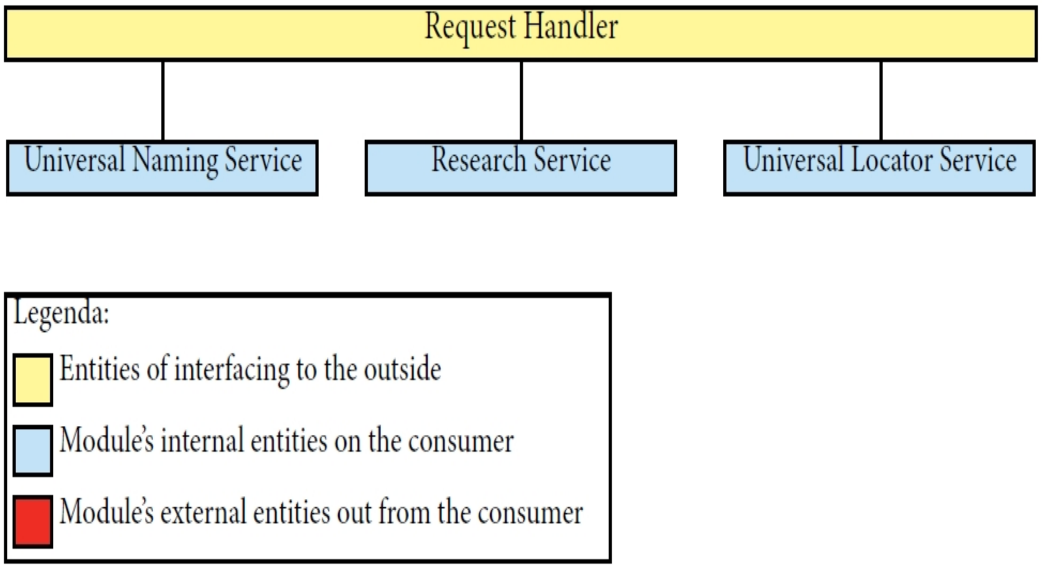

Request Handler
Universal Naming Service
Research Service
Universal Locator Service
Legenda:
🟨 Entities of interfacing to the outside
🟦 Module's internal entities on the consumer
🟥 Module's external entities out from the consumer

Figure 3.1: Main Architectural Model

As described above, a consumer can contain only a part of the module (for example, if it uses most the location service then this part of the module will be local on the consumer while the other could be distributed), and in this case the module is distributed, or the consumer can contain the entire module, so without any system distribution. In any case, the consumer is coordinated by the individual services so you do not have problems with latency and fault tolerance.

3.2 Naming Service Model

This module handles registrations and all claims relating to the *naming system*. As shown in the description in the following picture, also some components of this module's part can be distributed, so there is a further distribution of the model, this further distribution will be further detailed later in this section.

The infrastructural model of this module is based on the *three layer vertical modeling* and in one of these layers there is a further horizontal subdivision in many layers as you can see in the picture below.

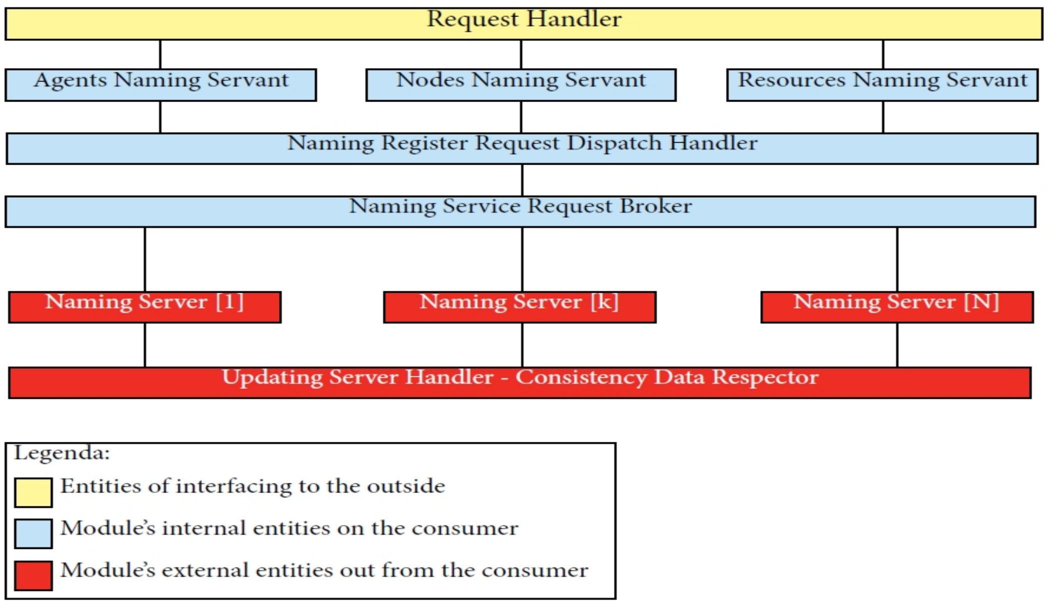

Request Handler
Agents Naming Servant
Nodes Naming Servant
Resources Naming Servant
Naming Register Request Dispatch Handler
Naming Service Request Broker
Naming Server [1]
Naming Server [k]
Naming Server [N]
Updating Server Handler - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.2: Naming Service Architectural Model

As seen in the picture there are the following three vertical layers:

- *outside interfacing layer*: this layer interface the module with the external consumer (for example the TuCSoN middleware)

- *internal request computation layer*: this layer is designed to compute the received requests from the above layer and return the results of the received requests (for example "OK" answer to a registration request received from an agent with a consequent return of an universal identifier for that agent)

- *data storage layer*: this layer is designed to manage the data storage relating to the naming service (for example the correspondence between agents and their universal identifiers)

In the following subsections will be described all details about the three above layers.

3.2.1 Outside Interfacing Layer

This layer is composed by a single entity: the **Request Handler** entity.

This entity has the aim to provide all primitives to the external cunsumers in order to allow them to use the *naming service* module. Logically, this entity provides all primitives to external consumers, then the **Request Handler** not only provides the *naming service primitives* but also all primitives about the *locator service* and the *research service*. As previously seen, the internal part of the *naming service* can be distributed over the network, so in this case, the external consumer does not perceive any effect caused by that because the **Request Handler** has also the aim to manage all those issues related to the components distribution.

The **Local Naming Service Outside Intefacing Layer's functions** can be also extended by using external modules because it has been studied in order to have an important extendibility, in fact, to extend this layer's functions a consumer has to comunicate with *Request Handler* using the provided primitives. This design has been studied in order to maintain the complete orthogonality of the *Naming Service Module*, consistently with the full designed module model (as described in the initial section of the full module in this chapter).

Starting from the previous considerations we have that this model appears suitable for use in multiple contexts, independently from the consumer (in the studied case the TuCSoN middleware).

Then using local or distributed structural policy, the model can be described in the following two pictures.

As you can see in the previous picture, there are two possible kind of *naming service outside interfacing layer*: the local one and the distributed

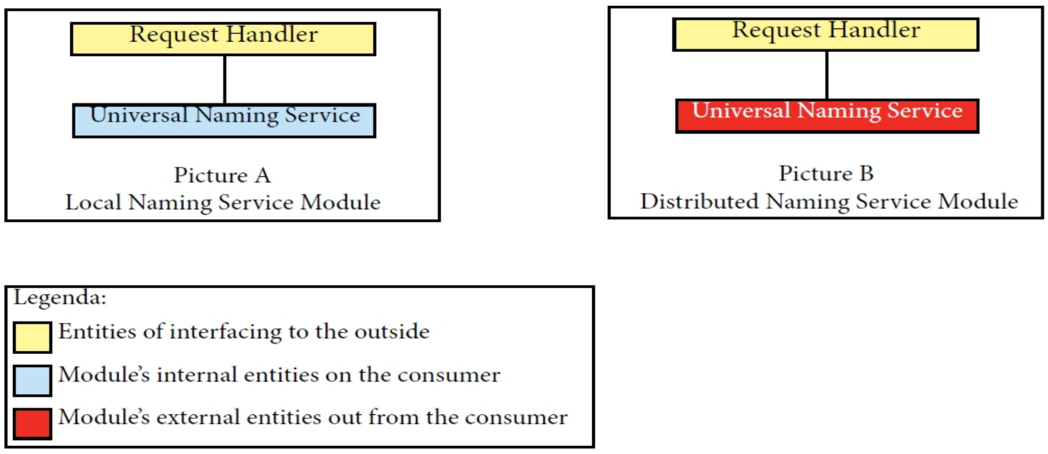

Request Handler
Universal Naming Service
Picture A
Local Naming Service Module
Request Handler
Universal Naming Service
Picture B
Distributed Naming Service Module
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.3: Local - Distributed Naming Service Main Architectural Model

one.

The Picture A shows the **Local Naming Service Outside Intefacing Layer** (logically referred to the **Local Naming Service Model**) where the naming module is completely on the consumer, so in this case, the **Request Handler** entity is in the same place of the rest of the naming module and the interfacing layer doesn't have to manage the distributed issues previously described about the communication with the internal part of the naming module.

The Picture B shows the **Distributed Naming Service Outside Intefacing Layer** (logically referred to the **Distributed Naming Service Model**) where the naming module is not in the same place of the rest of naming module, in fact, while the **Request Handler** in located on the consumer, the rest of the *naming service module* is in another place in order to have a distributed architectural model of the system. Logically, in this case the interfacing layer has to manage the distributed issues about network communication with the internal part of the naming module, so this layer has to manage also all functions about the transmission and the fault tolerance of this system.

3.2.2 Internal Request Computation Layer

This layer is composed by many entities in order to divide the managing of all computational functions related to the *naming service*. This division has also been made to reduce the entities's complexity in order to obtain an architectural model more extensible because it's easier to extend the entities's functionalities when these have a lesser computation complexity.

This layer can be distributed like the previously described one because the managing of network communication is the aim of the upper layer (the **Local Naming Service Outside Intefacing Layer**).

The following picture described the architectural model of the **Internal Request Computation Layer**.

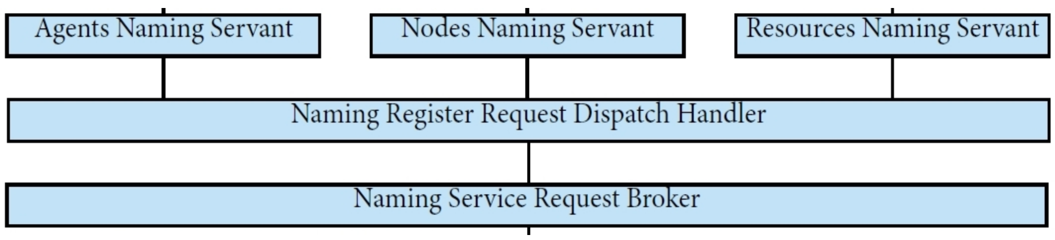

Agents Naming Servant
Nodes Naming Servant
Resources Naming Servant
Naming Register Request Dispatch Handler
Naming Service Request Broker

Figure 3.4: Internal Request Computation Naming Service Architectural Model

Starting from the previous picture we can describe all computational entities composing the internal module of the *naming service*. All of these entities will be described in relation to the presence of the TuCSoN middleware as naming service module's consumer. Then, these entities will be described in details in relation to the presence of *TuCSoN Agents*, *TuCSoN Nodes* and *Resources* (logically, resources are related to the *TuCSoN* model).

All this layer's entities are compatible with any consumer middleware because the only change to do would be related to the characteristics of the consumer entities.

The entities in the **Internal Request Computation Layer** are:

- **Naming Servant**: entity capable to serve each naming request (eg: create a new universal name for an agent, update description of an agent's name, etc.) received from *TuCSoN Agents*, *TuCSoN Nodes* and *Resources*. In details, there are the following specialized *Naming Servant* entities:

  - **Agents Naming Servant**: specialized *naming servant* entity capable to serve all request received from a *TuCSoN Agent*
  - **Nodes Naming Servant**: specialized *naming servant* entity capable to serve all request received from a *TuCSoN Node*
  - **Resources Naming Servant**: specialized *naming servant* entity capable to serve all request received from a *TuCSoN entity* that has to use a resource in a *TuCSoN system*; logically, this *TuCSoN system* has to be a consumer of the *naming service module* through *TuCSoN middleware*

- **Naming Register Request Dispatch Handler**: entity capable of dispatching request about *naming service* received from all servant entities (*Agents Naming Servant*, *Nodes Naming Servant* and *Resources Naming Servant*) in order to forward all these requests to the broker entity

- **Naming Service Request Broker**: entity capable of searching the current free server in order to serve request about *naming service* in order to manage all operations about *naming storaged data*. As you will see in the next chapter, this entity is called also from **Checker Entity** in order to check the existance of an *universal identifier* or in order to execute any other *naming service request*.

The **Naming Servant Entities** has been designed in order to have a *three tired model* because by adopting this design model these entities are less complex (it refers to the computational complexity) such as previously described and in order to obtain a better extensibility.

As previously said, this part of the *naming service module* can be distributed over the network or can be completely contained on the cunsumer. In the next paragraph will be described the local version of this module's part, so the situation where the **Internal Request Computation Layer** is completely contained on the consumer.

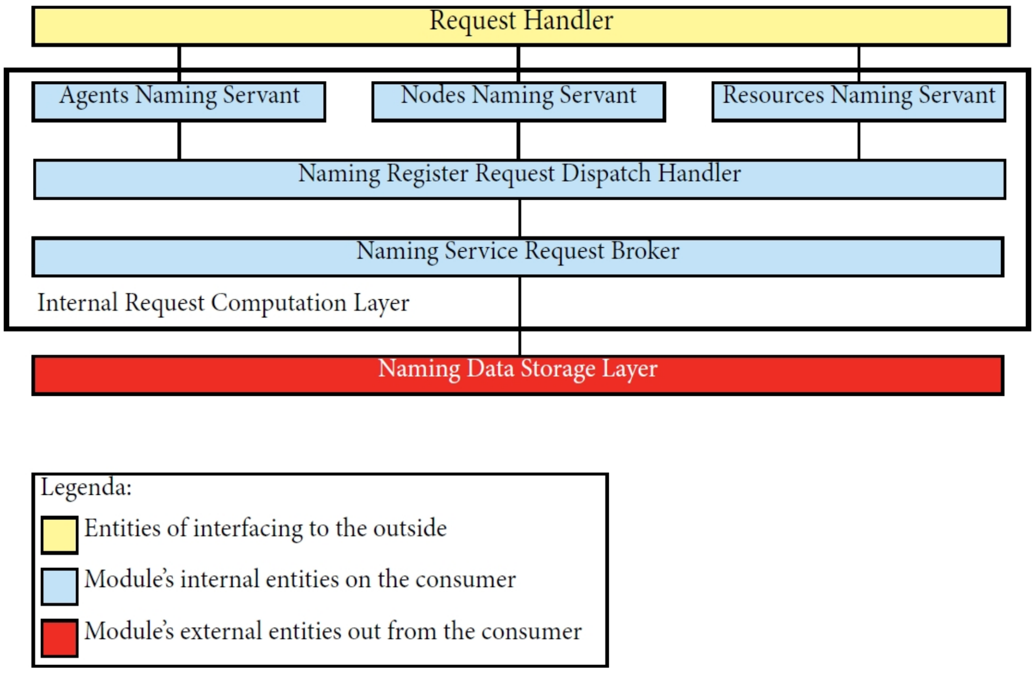

Request Handler
Agents Naming Servant
Nodes Naming Servant
Resources Naming Servant
Naming Register Request Dispatch Handler
Naming Service Request Broker
Internal Request Computation Layer
Naming Data Storage Layer
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.5: Local Naming Service Internal Architectural Model

As you can see in the above picture, when this naming module's layer is contained on the consumer, this level is directly connected with the **Request Handler** that's contained on the consumer as described in the previous section. In this case, all naming service requests are directly sent from the **Request Handler** to the **Naming Servant** entities without any distributed transmission. So there is no problem related to the fault tolerance and communication problems. The managing of distribution issues is related to the communication between the **Naming Service Request Broker** and the lower layer, called **Data Storage Layer**.

In the next paragraph will be described the distributed version of this module's part, so the situation where the **Internal Request Computation Layer** is outside from consumer while the **Request Handler** is contained on the consumer.

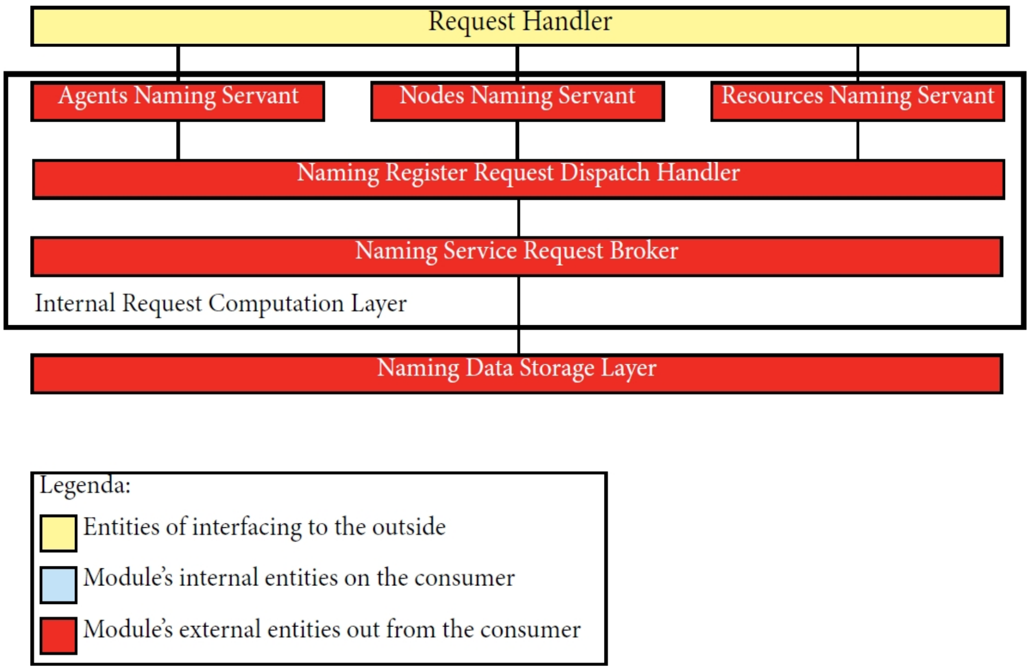

Request Handler
Agents Naming Servant
Nodes Naming Servant
Resources Naming Servant
Naming Register Request Dispatch Handler
Naming Service Request Broker
Internal Request Computation Layer
Naming Data Storage Layer
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.6: Distributed Naming Service Internal Architectural Model

As you can see in the above picture, in this case the naming module's layer is distributed over the network. In this case the **Internal Request Computation Layer** and the **Request Handler** are in two different places, in fact, while the **Request Handler** is cointained on the cunsumer, the **Internal Request Computation Layer** is cointained in a different place. These two layers comunicate through the connection between **Naming Servant** entities and the **Request Handler** entity. So there are two kind of distribution issues:

- distribution issues related to the communication between **Outside Interfacing Layer** and **Internal Request Computation Layer**
- distribution issues related to the communication between **Naming Service Request Broker** entity and the lower layer: the **Data Storage Layer**

The described distribution issues are managed by different entities, in fact in the firse case (**Outside Interfacing Layer - Internal Request Computation Layer**) the involved entities are the **Request Handler** entity and **Naming Servant** entities because these are their layers' borderline entities. Second (**Internal Request Computation Layer - Data Storage Layer**) the involved entities are the **Naming Service Request Broker** and corresponding entity in the lower layer because these are their layer's borderline entities.

All distribution management policies has to prevent and solve any problem about the fault tolerance and communication problems. So all these layers' borderline entities are very important for the efficiency in network distribution because if these mechanisms are not efficient there are problems related to the communicational and computational responsiveness and reliability of the complete *naming system* module.

The naming system module's lower layer and all its borderline entities will be described in the following section.

3.2.3 Data Storage Layer

This layer has the aim to manage the data storage and provide all functions about the *naming service data storage*.

This layer has been studied in order to have a better orthogonality between the naming system and the data storage system. In fact, the server distribution is completely independent from the naming system and the naming module.

The distributed architectural model about this layer has been studied in order to allow to manage data servers and their technology details independently from the *naming service module*, then, if the data servers' technology will be modified there are no changes to do about the *naming system* module.

A **naming server** can be located on a TuCSoN **node** (related to TuCSoN **middleware** as external consumer) or in another place; in fact, as you will se in following chapters, the **Naming Service Request Broker** saw in the previous section has the aim to manage all distribution issues related to the **Data Storage Layer**. All these distribution issues are also managed by a particular entity contained in this layer that will be described in the following paragraph (after the model picture).

The following picture described the architectural model of the **Data Storage Layer**.

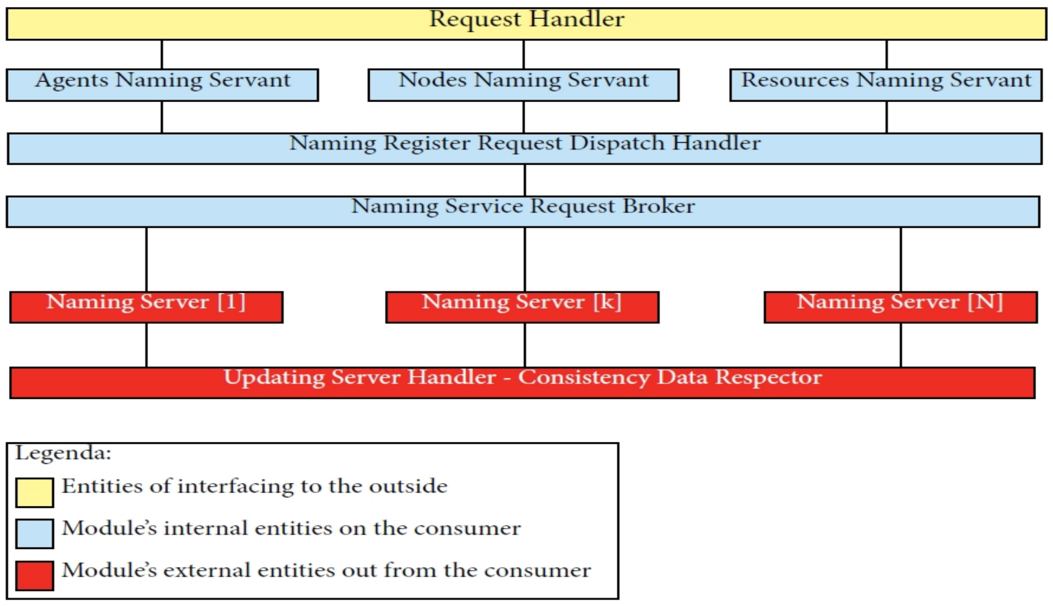

Request Handler
Agents Naming Servant
Nodes Naming Servant
Resources Naming Servant
Naming Register Request Dispatch Handler
Naming Service Request Broker
Naming Server [1]
Naming Server [k]
Naming Server [N]
Updating Server Handler - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.7: Data Storage Naming Service Internal Architectural Model

The entities in the **Data Storage Layer** are:

- **Naming Server**: entity capable of storing data about names and references about Agents, Nodes and Resources's universal identifiers
- **Updating Server Handler - Consistency Data Respector**: entity capable of updating data about *Naming Servers* in order to have consistent updates across all distributed servers in order to achieve consistency in saved data; this entity also manage all the distribution issues about the **Naming Servers** because it has the aim to check the connectivity of distributed servers and their state

The broker architectural model related to the communication between the **Internal Request Computation Layer** and the **Data Storage Layer** has been studied in order to obtain a better efficency because the latency will be lesser then other architectural models.

An excellent solution for the network distribution of *Naming Servers* is **Cassandra** (described in the previous chapter) that provides all necessary services in order to have a good data storage distribution.

In this case, the *Coordinator* role is set to the *Updating Server Handler* because this is the entity able to manage and administrate all issues about the *Data Consistency* (in fact, this entity is described as *Consistency Data Respector*).

This role is very important for the correct execution of all functions about the **Data Storage Layer** because it is the *data storage's core.*

There is also another motivation for this choice about **Cassandra**; this motivation is about the future evolution about the *data storage system* because, after specific studies, this model results completly independent from the considered module, in fact, if **Cassandra** will be updated maintaining all its interfacing primitives, the main infrastructural model won't be modified because it's not part of **Cassandra** (in fact, in this case study this module is a consumer for **Cassandra**).

3.3 Locator Service Model

This module handles registrations and all claims relating to the *locator system*. As shown in the legend in the following picture, also some components of this module's part can be distributed, so there is a further distribution of the model, this further distribution will be further detailed later in this section. The infrastructural model of this module is based on the *three layer vertical modeling* and in one of these layers there is a further horizontal subdivision in many layers as you can see in the picture below.

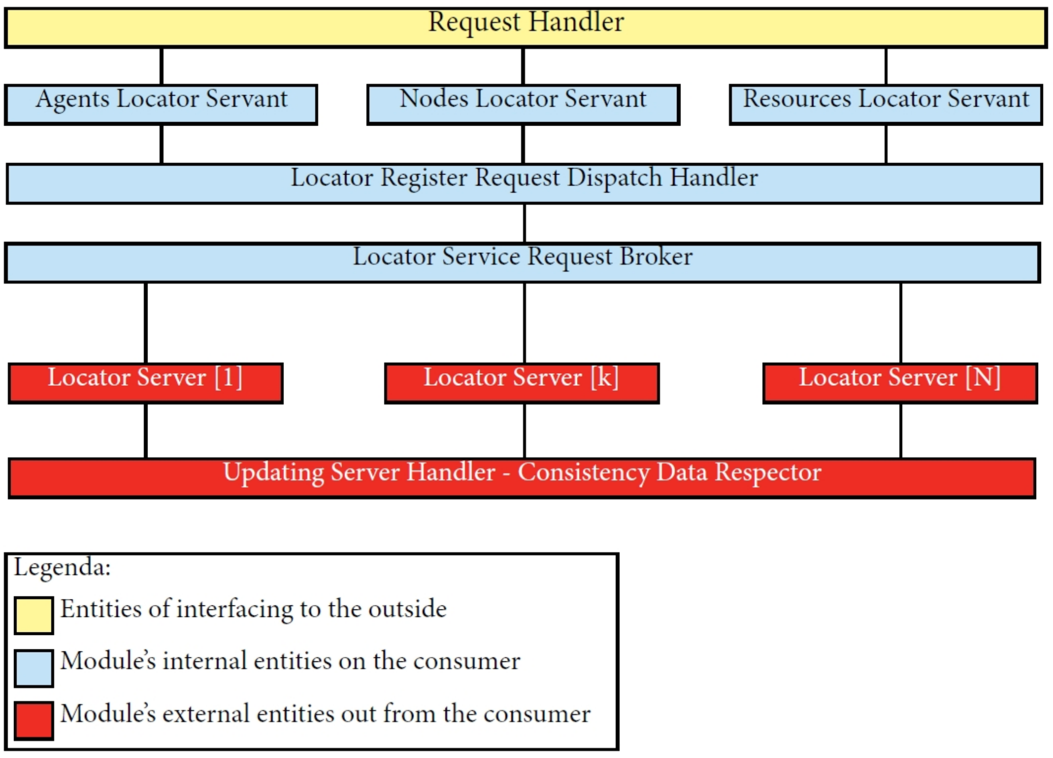

Request Handler
Agents Locator Servant
Nodes Locator Servant
Resources Locator Servant
Locator Register Request Dispatch Handler
Locator Service Request Broker
Locator Server [1]
Locator Server [k]
Locator Server [N]
Updating Server Handler - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.8: Locator Service Architectural Model

As seen in the picture there are the following three vertical layers:

- *outside interfacing layer*: this layer interface the module with the external consumer (for example the TuCSoN middleware)

- *internal request computation layer*: this layer is designed to compute the received requests from the above layer and return the results of the received requests (for example "OK" answer to a location registration request received from an agent)
- *data storage layer*: this layer is designed to manage the data storage relating to the locator service (for example the correspondence between agents and their current location)

In the following subsections will be described all details about the three above layers.

3.3.1 Outside Interfacing Layer

This layer is composed by a single entity: the **Request Handler** entity.

This entity has the aim to provide all primitives to the external cunsumers in order to allow them to use the *locator service* module. Logically, this entity provides all primitives to external consumers, then the **Request Handler** not only provides the *locator service primitives* but also all primitives about the *naming service* (seen in the previous section) and the *research service*. As previously seen, the internal part of the *locator service* can be distributed over the network, so in this case, the external consumer does not perceive any effect caused by that because the **Request Handler** has also the aim to manage all those issues related to the components distribution.

The **Local Locator Service Outside Intefacing Layer's functions** can be also extended by using external modules because it has been studied in order to have an important extendibility, in fact, to extend this layer's functions a consumer has to comunicate with *Request Handler* using the provided primitives. This design has been studied in order to maintain the complete orthogonality of the *Locator Service Module*, consistently with the full designed module model (as described in the initial section of the full module in this chapter).

Starting from the previous considerations we have that this model appears suitable for use in multiple contexts, independently from the consumer (in the studied case the TuCSoN middleware).

Then using local or distributed structural policy, the model can be described in the following two pictures.

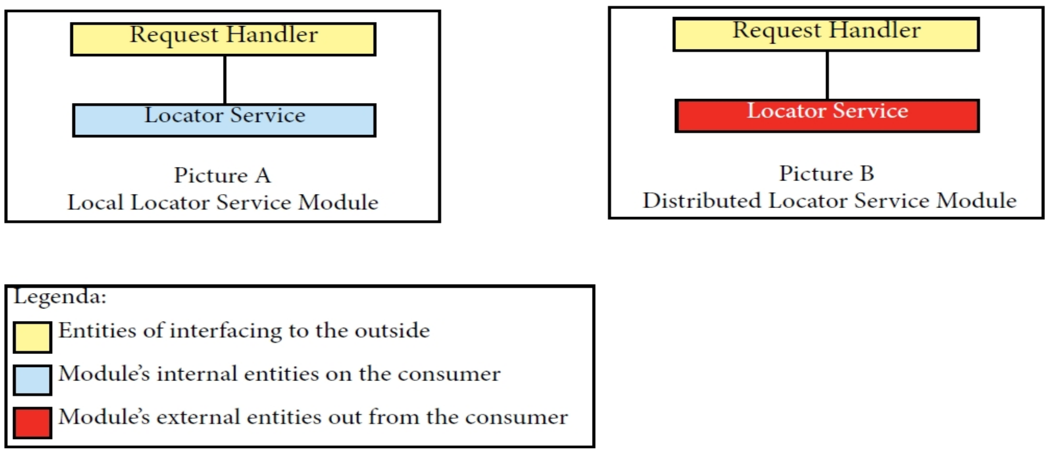

Request Handler
Locator Service
Picture A
Local Locator Service Module
Request Handler
Locator Service
Picture B
Distributed Locator Service Module
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.9: Local - Distributed Locator Service Main Architectural Model

As you can see in the previous picture, there are two possible kind of *locator service outside interfacing layer*: the local one and the distributed one.

The Picture A shows the **Local Locator Service Outside Intefacing Layer** (logically referred to the **Local Locator Service Model**) where the locator module is completely on the consumer, so in this case, the **Request Handler** entity is in the same place of the rest of the locator module and the interfacing layer doesn't have to manage the distributed issues previously described about the communication with the internal part of the locator module.

The Picture B shows the **Distributed Locator Service Outside Intefacing Layer** (logically referred to the **Distributed Locator Service Model**) where the locator module is not in the same place of the rest of locator module, in fact, while the **Request Handler** in located on the consumer, the rest of the *locator service module* is in another place in order to have a distributed architectural model of the system. Logically, in this case the interfacing layer has to manage the distributed issues about network communication with the internal part of the locator module, so this layer has to manage also all functions about the transmission and the fault tolerance of this system.

3.3.2 Internal Request Computation Layer

This layer is composed by many entities in order to divide the managing of all computational functions related to the *locator service*. This division has also been made to reduce the entities's complexity in order to obtain an architectural model more extensible because it's easier to extend the entities's functionalities when these have a lesser computation complexity.

This layer can be distributed like the previously described one because the managing of network communication is the aim of the upper layer (the **Local Locator Service Outside Intefacing Layer**).

The following picture describes the architectural model of the **Internal Request Computation Layer**.

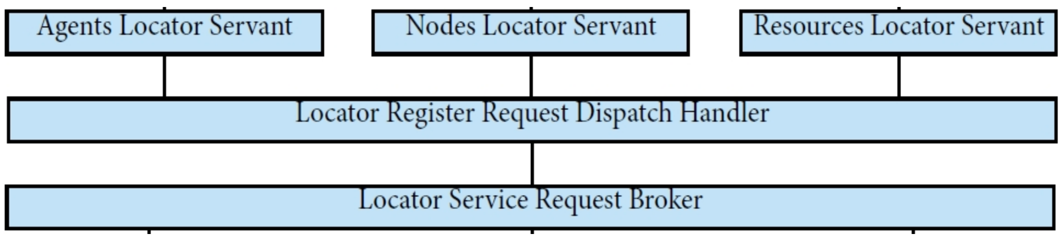

Agents Locator Servant
Nodes Locator Servant
Resources Locator Servant
Locator Register Request Dispatch Handler
Locator Service Request Broker

Figure 3.10: Internal Request Computation Locator Service Architectural Model

Starting from the previous picture we can describe all computational entities composing the internal module of the *locator service*. All of these entities will be described in relation to the presence of the TuCSoN middleware as locator service module's consumer. Then, these entities will be described in details in relation to the presence of *TuCSoN Agents*, *TuCSoN Nodes* and *Resources* (logically, resources are related to the *TuCSoN* model).

All this layer's entities are compatible with any consumer middleware because the only change to do would be related to the characteristics of the consumer entities.

The entities in the **Internal Request Computation Layer** are:

- **Locator Servant**: entity capable to serve each locator request (eg: register the current location of an agent, update the location of an agent, etc.) received from *TuCSoN Agents*, *TuCSoN Nodes* and *Resources*. In details, there are the following specialized *Locator Servant* entities:

  - **Agents Locator Servant**: specialized *locator servant* entity capable to serve all request received from a *TuCSoN Agent*
  - **Nodes Locator Servant**: specialized *locator servant* entity capable to serve all request received from a *TuCSoN Node*
  - **Resources Locator Servant**: specialized *locator servant* entity capable to serve all request received from a *TuCSoN entity* that has to use a resource in a *TuCSoN system*; logically, this *TuCSoN system* has to be a consumer of the *naming service module* through *TuCSoN middleware*

- **Locator Register Request Dispatch Handler**: entity capable of dispatching request about *locator service* received from all servant entities (*Agents Locator Servant*, *Nodes Locator Servant* and *Resources Locator Servant*) in order to forward all these requests to the broker entity

- **Locator Service Request Broker**: entity capable of searching the current free server in order to serve request about *locator service* in order to manage all operations about *locator storaged data*. As you will see in the next chapter, this entity is called also from **Checker Entity** in order to check the *location* of an agent or a node or in order to execute any other *locator service request*.

The **Locator Servant Entities** has been designed in order to have a *three tired model* because by adopting this design model these entities are less complex (it refers to the computational complexity) such previously described and in order to obtain a better extensibility.

As previously said, this part of the *locator service module* can be distributed over the network or can be completely contained on the cunsumer. In the next paragraph will be described the local version of this module's part, so the situation where the **Internal Request Computation Layer** is completely contained on the consumer.

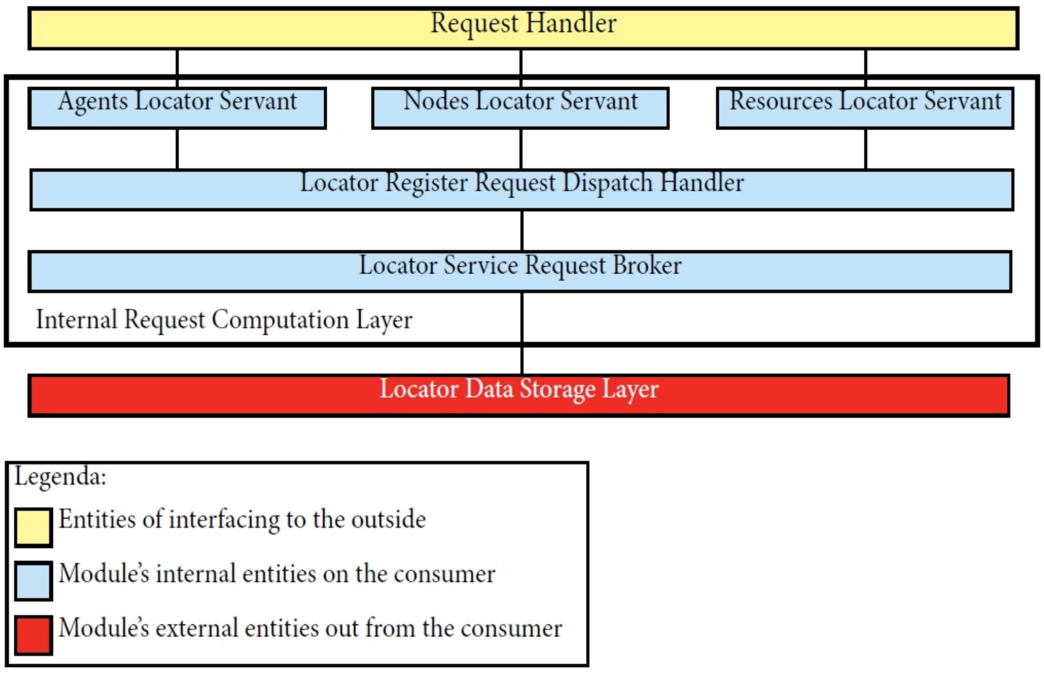

Request Handler
Agents Locator Servant
Nodes Locator Servant
Resources Locator Servant
Locator Register Request Dispatch Handler
Locator Service Request Broker
Internal Request Computation Layer
Locator Data Storage Layer
Legenda:
🟨 Entities of interfacing to the outside
🟦 Module's internal entities on the consumer
🟥 Module's external entities out from the consumer

Figure 3.11: Local Locator Service Internal Architectural Model

As you can see in the above picture, when this locator module's layer is contained on the consumer, this level is directly connected with the **Request Handler** that's contained on the consumer as described in the previous section. In this case, all locator service requests are directly sent from the **Request Handler** to the **Locator Servant** entities without any distributed transmission. So there is no problem related to the fault tolerance and communication problems. The managing of distribution issues is related to the communication between the **Locator Service Request Broker** and the lower layer, called **Data Storage Layer**.

In the next paragraph will be described the distributed version of this module's part, so the situation where the **Internal Request Computation Layer** is outside from consumer while the **Request Handler** is contained on the consumer.

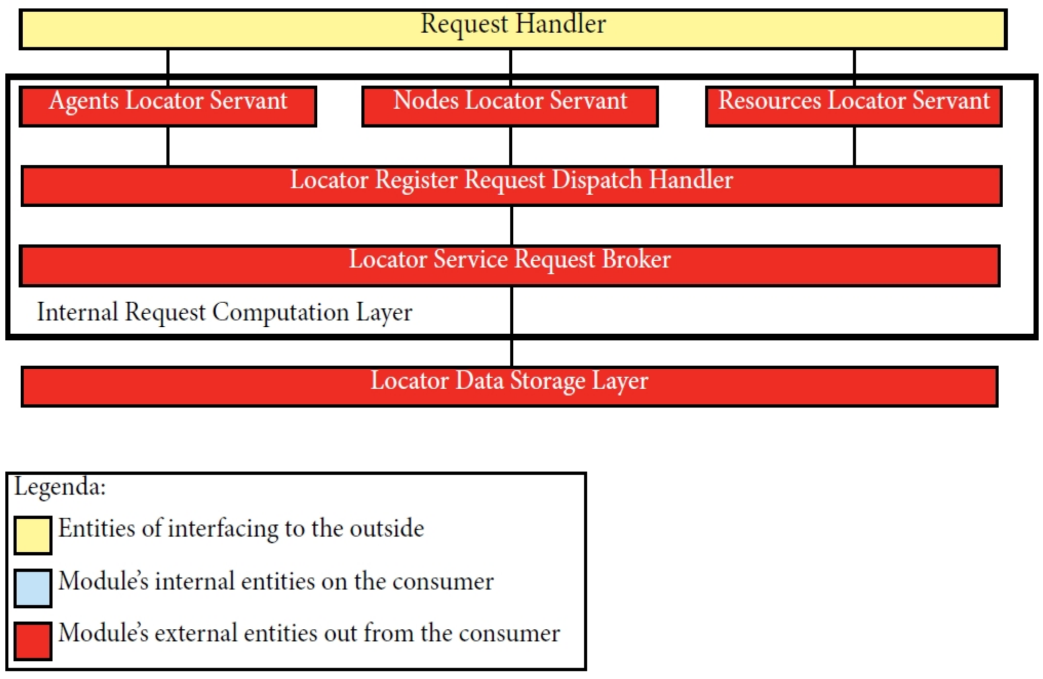

Request Handler
Agents Locator Servant
Nodes Locator Servant
Resources Locator Servant
Locator Register Request Dispatch Handler
Locator Service Request Broker
Internal Request Computation Layer
Locator Data Storage Layer
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.12: Distributed Locator Service Internal Architectural Model

As you can see in the above picture, in this case the locator module's layer is distributed over the network. In this case the **Internal Request Computation Layer** and the **Request Handler** are in two different places, in fact, while the **Request Handler** is cointained on the cunsumer, the **Internal Request Computation Layer** is cointained in a different place. These two layers comunicate through the connection between **Locator Servant** entities and the **Request Handler** entity. So there are two kind of distribution issues:

- distribution issues related to the communication between **Outside Interfacing Layer** and **Internal Request Computation Layer**
- distribution issues related to the communication between **Locator Service Request Broker** entity and the lower layer: the **Data Storage Layer**

The described distribution issues are managed by different entities, in fact in the firse case (**Outside Interfacing Layer - Internal Request Computation Layer**) the involved entities are the **Request Handler** entity and **Locator Servant** entities because these are their layers' borderline entities. Second (**Internal Request Computation Layer - Data Storage Layer**) the involved entities are the **Locator Service Request Broker** and corresponding entity in the lower layer because these are their layer's borderline entities.

All distribution management policies have to prevent and solve any problem about the fault tolerance and communication problems. So all these layers' borderline entities are very important for the efficiency in network distribution because if these mechanisms are not efficient there are problems related to the communicational and computational responsiveness and reliability of the complete *locator system* module.

The locator system module's lower layer and all its borderline entities will be described in the following section.

3.3.3 Data Storage Layer

This layer has the aim to manage the data storage and provide all functions about the *locator service data storage.*

This layer has been studied in order to have a better orthogonality between the locator system and the data storage system. In fact, the server distribution is completely independent from the locator system and the locator module.

The distributed architectural model about this layer has been studied in order to allow to manage data servers and their technology details independently from the *locator service module*, then, if the data servers' technology will be modified there are no changes to do about the *locator system* module.

A **locator server** can be located on a TuCSoN **node** (related to TuCSoN **middleware** as external consumer) or in another place; in fact, as you will se in following chapters, the **Locator Service Request Broker** saw in the previous section has the aim to manage all distribution issues related to the **Data Storage Layer**. All these distribution issues are also managed by a particular entity contained in this layer that will be described in the following paragraph (after the model picture).

The following picture described the architectural model of the **Data**

Storage Layer.

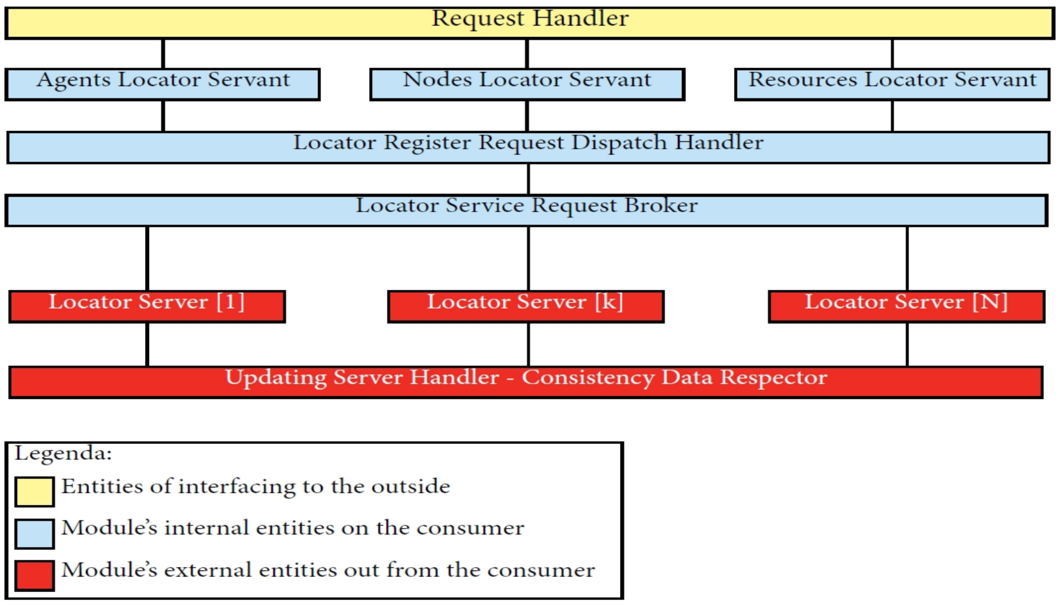

Request Handler
Agents Locator Servant
Nodes Locator Servant
Resources Locator Servant
Locator Register Request Dispatch Handler
Locator Service Request Broker
Locator Server [1]
Locator Server [k]
Locator Server [N]
Updating Server Handler - Consistency Data Respector
Legenda:
🟨 Entities of interfacing to the outside
🟦 Module's internal entities on the consumer
🟥 Module's external entities out from the consumer

Figure 3.13: Data Storage Locator Service Internal Architectural Model

The entities in the **Data Storage Layer** are:

- **Naming Server**: entity capable of storing data about names and references about Agents, Nodes and Resources's current location

- **Updating Server Handler - Consistency Data Respector**: entity capable of updating data about *Locator Servers* in order to have consistent updates across all distributed servers in order to achieve consistency in saved data; this entity also manage all the distribution issues about the **Locator Servers** because it has the aim to check the connectivity of distributed servers and their state

The broker architectural model related to the communication between the **Internal Request Computation Layer** and the **Data Storage Layer** has been studied in order to obtain a better efficency because the latency will be lesser then other architectural models.

An excellent solution for the over network distribution of *Naming Servers* is **Cassandra** (described in the previous chapter) that provides all necessary services in order to have a good data storage distribution.

In this case, the *Coordinator* role is set to the *Updating Server Handler* because this is the entity able to manage and administrate all issues about the *Data Consistency* (in fact, this entity is described as *Consistency Data Respector*).

This role is very important for the correct execution of all functions about the **Data Storage Layer** because it is the *data storage's core.*

There is also another motivation for this choice about **Cassandra**; this motivation is about the future evolution about the *data storage system* because, after specific studies, this model results completly independent from the considered module, in fact, if **Cassandra** will be updated maintaining all its interfacing primitives, the main infrastructural model won't be modified because it's not part of **Cassandra** (in fact, in this case study this module is a consumer for **Cassandra**).

3.4 Research Service Model

This module handles all claims relating to the research and checking functions about the *naming module* and the *locator module*. Like other seen modules, also this module's internal part can be distributed as will be described in following section. Unlike other modules, this module's infrastructural model is not based on the three layer vertical modeling but it's based on the *two layer vertical modeling* and, unlike others two modules, there is no module with a further orizzontal subdivision, as you can see in the picture below.

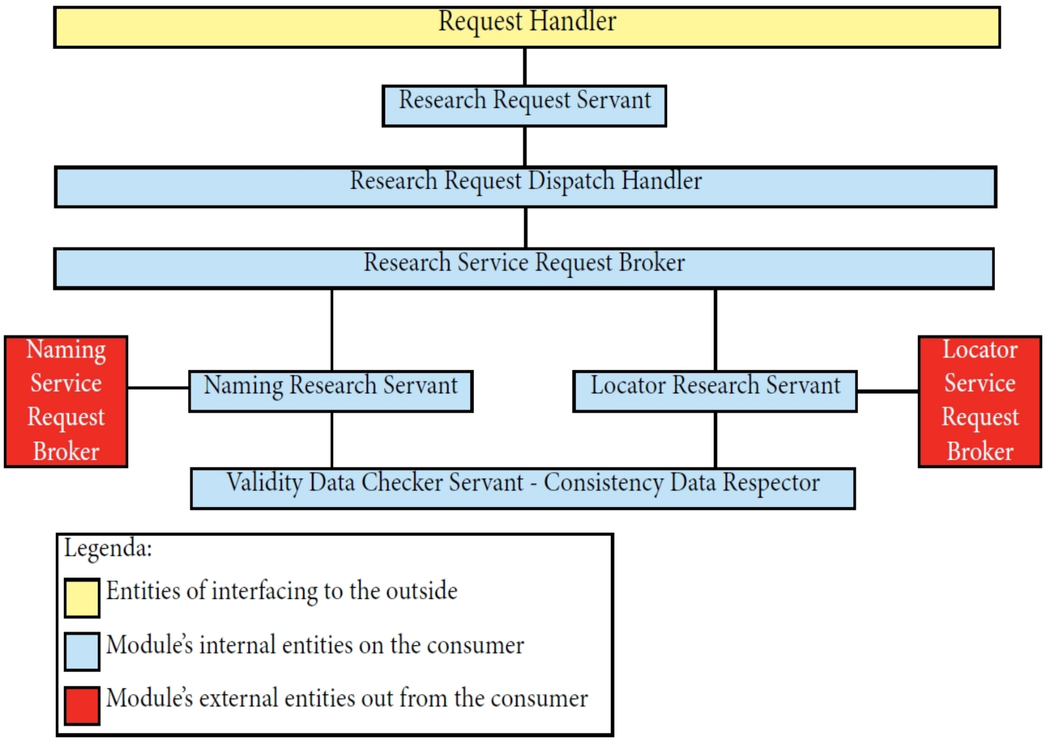

Request Handler
Research Request Servant
Research Request Dispatch Handler
Research Service Request Broker
Naming Service Request Broker
Naming Research Servant
Locator Research Servant
Locator Service Request Broker
Validity Data Checker Servant - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.14: Research Service Architectural Model

As seen in the picture, there are the following two vertical layers:

- *outside interfacing layer*: this layer interface the module with the external consumer (for example the TuCSoN middleware)

- *internal request computation layer*: this layer is designed to compute the received requests from the above layer and return the result of the received requests(for example an agent's current location)

In the following subsections will be described all details about these two layer above introduced.

3.4.1 Outside Interfacing Layer

This layer is composed by a single entity: the **Request Handler** entity.

This entity has the aim to provide all primitives to the external cunsumers in order to allow them to use the *research service* module. Logically, this entity provides all primitives to external consumers, then the **Request Handler** not only provides the *research service primitives* but also all primitives about the *naming service* and the *locator service* (both seen in the previous sections). As previously seen, the internal part of the *research service* can be distributed over the network, so in this case, the external consumer does not perceive any effect caused by that because the **Request Handler** has also the aim to manage all those issues related to the components distribution.

The **Local Research Service Outside Intefacing Layer's functions** can be also extended by using external modules because it has been studied in order to have an important extendibility, in fact, to extend this layer's functions a consumer has to comunicate with *Request Handler* using the provided primitives. This design has been studied in order to maintain the complete orthogonality of the *Research Service Module*, consistently with the full designed module model (as described in the initial section of the full module in this chapter).

Starting from the previous considerations we have that this model appears suitable for use in multiple contexts, independently from the consumer (in the studied case the TuCSoN middleware).

Then using local or distributed structural policy, the model can be described in the following two pictures.

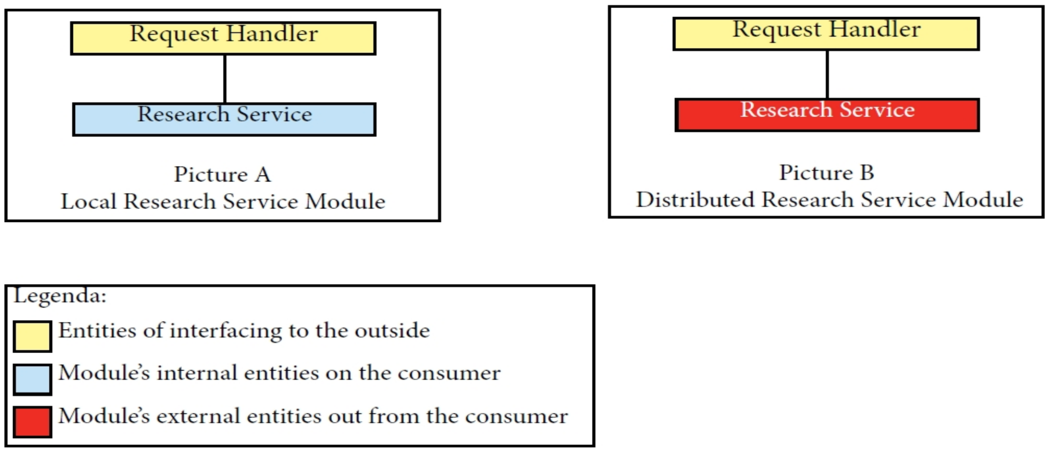

Request Handler
Research Service
Picture A
Local Research Service Module
Request Handler
Research Service
Picture B
Distributed Research Service Module
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.15: Local - Distributed Research Service Main Architectural Model

As you can see in the previous picture, there are two possible kind of *research service outside interfacing layer*: the local one and the distributed one.

The Picture A shows the **Local Research Service Outside Intefacing Layer** (logically referred to the **Local Research Service Model**) where the research module is completely on the consumer, so in this case, the **Request Handler** entity is in the same place of the rest of the locator module and the interfacing layer doesn't have to manage the distributed issues previously described about the communication with the internal part of the locator module.

The Picture B shows the **Distributed Research Service Outside Intefacing Layer** (logically referred to the **Distributed Research Service Model**) where the research module is not in the same place of the rest of locator module, in fact, while the **Request Handler** in located on the consumer, the rest of the *research service module* is in another place in order to have a distributed architectural model of the system. Logically, in this case the interfacing layer has to manage the distributed issues about network communication with the internal part of the locator module, so this layer has to manage also all functions about the transmission and the fault tolerance of this system.

3.4.2 Internal Request Computation Layer

This layer is composed by many entities in order to divide the managing of all computational functions related to the *research service*. This division has also been made to reduce the entities's complexity in order to obtain an architectural model more extensible because it's easier to extend the entities's functionalities when these have a lesser computation complexity.

This layer can be distributed like the previously described one because the managing of network communication is the aim of the upper layer (the **Local Research Service Outside Intefacing Layer**).

The following picture described the architectural model of the **Internal Request Computation Layer**.

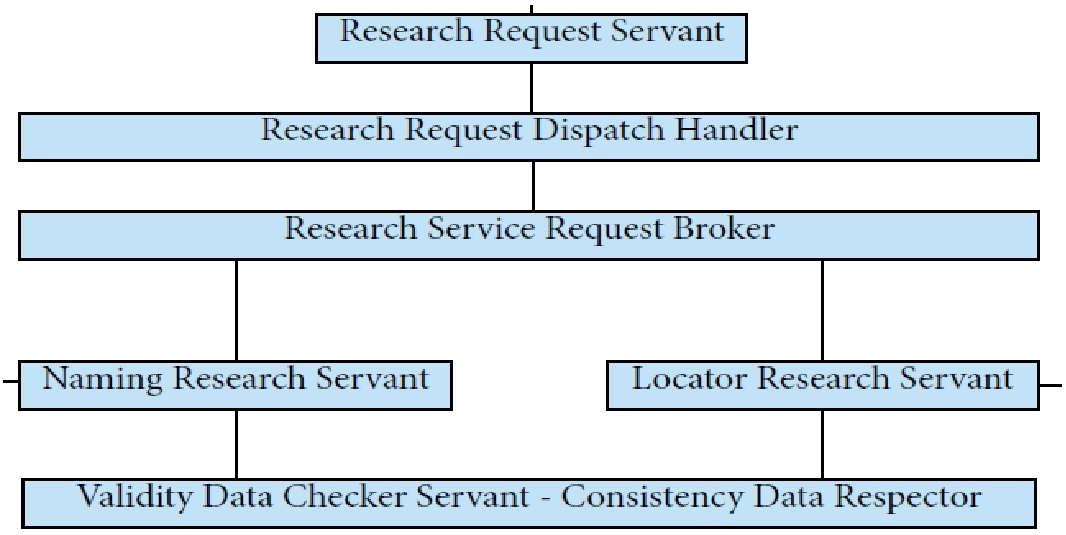

Research Request Servant
Research Request Dispatch Handler
Research Service Request Broker
Naming Research Servant
Locator Research Servant
Validity Data Checker Servant - Consistency Data Respector

Figure 3.16: Internal Request Computation Research Service Architectural Model

Starting from the previous picture we can describe all computational entities composing the internal module of the *research service*. All of these entities will be described in relation to the presence of the TuCSoN middleware as research service module's consumer. Then, these entities will be described in details in relation to the presence of *TuCSoN Agents*, *TuCSoN Nodes* and *Resources* (logically, resources are related to the *TuCSoN* model).

All this layer's entities are compatible with any consumer middleware because the only change to do would be related to the characteristics of the consumer entities.

The entities in the **Internal Request Computation Layer** are:

- **Research Request Servant**: entity capable to receive all search requests in order to return the name or the location or any Naming and Locator services' data about an Agent, Node or a Resource; this entity is also responsable to check if the received request is a valid request or not (for example if the request's syntax is correct)

- **Research Request Dispatch Handler**: entity able to dispatch any received search request in order to check its validity and send it to the broker entity (for example check if the received request has as target the *naming research service* or the *locator research service* in order to refuse all unknown request in terms of unknown type target because this module can receive only request about *naming research service* and *locator research service*)

- **Research Service Request Broker**: entity able to forward the request to the specific servant entity after the received request type checking (for example after the check if the received request is a naming or a locator service request)

- **Naming Research Servant**: entity able to compute any naming service request from the external consumer (for example to check if an universal identifier is related to an TuCSoN agent or a TuCSoN node or to get the universal identifier about a TuCSoN agent); this entity communicate with the **Naming Service Request Broker** and forward the request to that entity in order to get the request result and send it to upper layers, so to reply to external consumer

- **Locator Research Servant**: entity able to compute any locator service request from the external consumer (for example to get the current location of a TuCSoN agent or a TuCSoN node); this entity communicate with the **Locator Service Request Broker** and forward the request to that entity in order to get the request result and send it to upper layers, so to reply to external consumer

- **Validity Data Checker Servant - Consistency Data Respector**: entity capable to check if the result data from the **Research Servant** entities are correct; so this entity has the aim to mantain data consistency between storaged data in other modules in relation with the validity of the received research requests (for example if a locator research request is received, this entity has to check if the related TuCSoN entity - an agent, a node or a resource - specified in the request is really an agent, a node or a resource, relating to the request's target entity)

As previously said, this part of the *research service module* can be distributed over the network or can be completely contained on the cunsumer. In the next paragraph will be described the local version of this module's part, so the situation where the **Internal Request Computation Layer** is completely contained on the consumer.

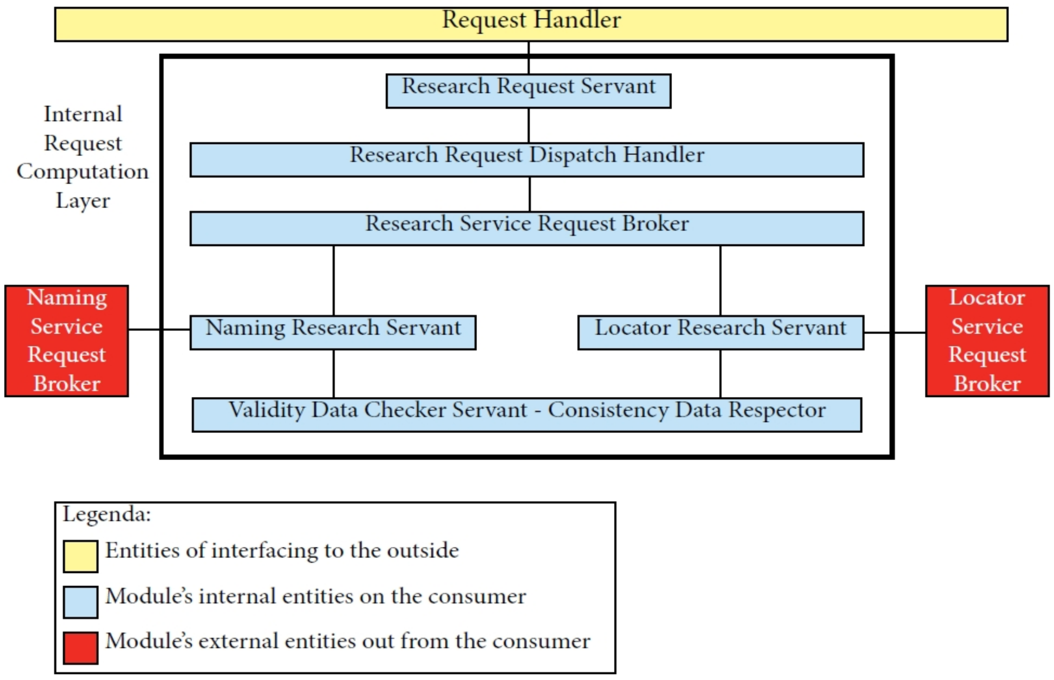

Request Handler
Internal Request Computation Layer
Research Request Servant
Research Request Dispatch Handler
Research Service Request Broker
Naming Service Request Broker
Naming Research Servant
Locator Research Servant
Locator Service Request Broker
Validity Data Checker Servant - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.17: Local Research Service Internal Architectural Model

As you can see in the above picture, when this locator module's layer is contained on the consumer, this level is directly connected with the **Request Handler** that's contained on the consumer as described in the previous section. In this case, all reseach service requests are directly sent from the **Request Handler** to the **Research Request Servant** entity without any distributed transmission. So there is no problem related to the fault tolerance and communication problems. The managing of distribution issues is related to the communication between the **Naming Request Servant** and the relative broker entity, the **Naming Service Request Broker**, only if that entity is distributed because if the **Naming Service Module** is on the consumer too, there are no problems related to network distribution. There is the same situation about the other servant, in fact, if the **Locator Research Servant** is placed on the consumer and the **Locator Service Request Broker** is not on it, there are issues relating to network distribution, while there are no issues about distribution if the **Locator Service Module** is on the consumer too, because they will be directly connected.

In the next paragraph will be described the distributed version of this module's part, so the situation where the **Internal Request Computation Layer** is outside from consumer while the **Request Handler** is contained on the consumer.

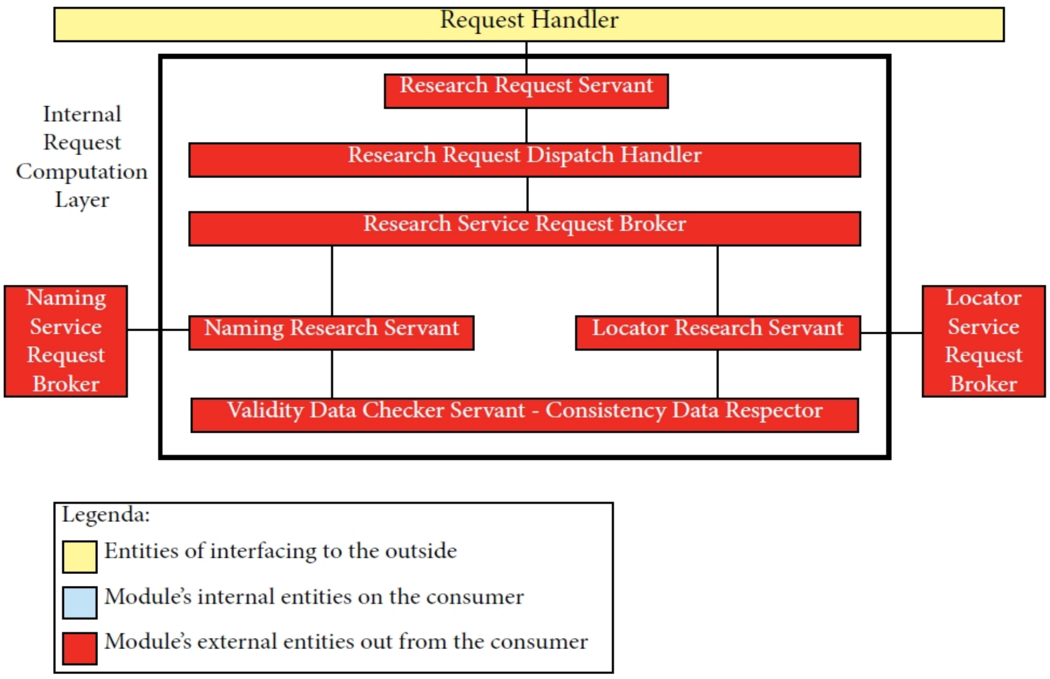

Request Handler
Internal Request Computation Layer
Research Request Servant
Research Request Dispatch Handler
Research Service Request Broker
Naming Service Request Broker
Naming Research Servant
Locator Research Servant
Locator Service Request Broker
Validity Data Checker Servant - Consistency Data Respector
Legenda:
Entities of interfacing to the outside
Module's internal entities on the consumer
Module's external entities out from the consumer

Figure 3.18: Distributed Research Service Internal Architectural Model

As you can see in the above picture, in this case the locator module's layer is distributed over the network. In this case the **Internal Request Computation Layer** and the **Request Handler** are in two different places, in fact, while the **Request Handler** is cointained on the cusumer, the **Internal Request Computation Layer** is cointained in a different place. These two layers comunicate through the connection between **Research Request Servant** entity and the **Request Handler** entity. So there are three kind of distribution issues:

- distribution issues related to the communication between **Outside Interfacing Layer** and **Internal Request Computation Layer**
- distribution issues related to the communication between **Naming Research Servant** entity and the **Naming Service Request Broker**

- distribution issues related to the communication between **Locator Research Servant** entity and the **Locator Service Request Broker**

The described distribution issues are managed by different entities, in fact in the firse case (**Outside Interfacing Layer - Internal Request Computation Layer**) the involved entities are the **Request Handler** entity and **Research Request Servant** entities because these are their layers' borderline entities.

Second (**Naming Research Servant - Naming Service Request Broker**) the involved entities are the **Naming Research Servant** and the **Naming Service Request Broker** because they are distributed over network and these entities are the borderline's entities about the **Internal Request Computation Layer** related to each module: **Research Service Module** and **Naming Service Module**.

In the end, in the last case (**Locator Research Servant - Locator Service Request Broker**) the involved entities are the **Locator Research Servant** and the **Locator Service Request Broker** because they are distributed over network and these entities are the borderline's entities about the **Internal Request Computation Layer** related to each module: **Research Service Module** and **Locator Service Module**.

This distributed model is very important because, with the possible distribution of the other module's **Internal Request Computation Layer**, it's possible to implement the full module in many ways, so it can be implemented with the distributed model or not depending on the case (for example if an agent is on a mobile device, the distributed model for the full module will be better because the computational load about it will be lesser, while if an agent is on a powerfull pc, the local model for the full module will be better because that machine's computational power can sadisfy the full module's computational load without problems).

However, all distribution management policies have to prevent and solve any problem about the fault tolerance and communication problems. So all borderline entities previously seen are very important for the efficiency in network distribution because if these mechanisms are not efficient there are problems related to the communicational and computational responsiveness and reliability of the complete *research system* module.

Chapter 4

Universal Naming Syntax

In this chapter a new syntax will be introduced for the **Universal Naming System** used in the module seen in the previous chapter.

This syntax has been studied in order to obtain a new way to give an **UID** (stand for **Universal IDentifier**) to a consumer entity of this module. This choice is derived from the technology evolution of the network distribution and the mobility of all entities associated with any device (for example if an entity is located on a smartphone, this entity will be mobile because there are many way to obtain a network connection associating to the device mobility).

This new naming definition has been needed in order to solve many problems and issues derived from the technology evolution of machines and device, first of all the *mobility* characteristic that is a new foundamental property of the modern devices and machines. These issues are related to many middleware designed before the *mobility technologic evolution* like TuCSoN , the consumer middleware considered in this study for the described module.

TuCSoN has been designed when there was only the *machine concept* about the network communication as described in the first chapter in the related naming section, so the only need was about the *static network naming* like the syntax previously seen [***NetworkID : PortNumber***] because a machine was identified in the network with only these two values. With the introduction of the *device concept* was born the *mobility property* of a device or a machine, that can move in the space so its old naming system version identifier is out of date because in many cases it is not valid (for example if an agent on a device change its location because the device it's moving, the agent's *ID* changes!) because the **identifier** must be absolute during all lifecycle of an entity.

Another foundamental evolution is the introduction of the *Cloud Computing*, a variety of different types of computing concepts that involve a large number of computers connected through a real-time communication network [34]. The *Cloud Computing* is involved in all modern technologies and middlewares, so it's important to improve the efficiency of the *naming system* in order to obtain an **universal unique identifier** valid in all modern situations described.

Nowadays many middlewares entities can be located on the *Cloud* (for example in TuCSoN an agent can be located on the *Cloud*) so, it was decided to study and design a new *naming system* that allows to give a valid **unique universal identifier** to an entity when it is located on a mobile device, on a machine or on the cloud. This need derive from the *mobility issues* relating to the *identifier validity* of a name given by the previous naming system because that naming system abstraction is not valid if an entity is located on the cloud because its *ID* changes and this situation is not admissible like the *mobile device* situation previously described.

This new *naming system syntax* has been studied based on the fact that TuCSoN is *programmed in Java* because this is the consumer middleware considerated in this study. It's important the reference to *Java* because, as will be described in the following section, this platform provides primitives that give an **unique universal identifier** for an entity; this identifier is valid in all crucial situations derived from the introduction of the *mobility property* of devices and machines. In fact, all problems and issues previously described about the old TuCSoN *naming system* are completely solved by this new system because this identifier is valid during all lifecycle of an entity.

Starting from the previous considerations and issues, the most important advantage of this syntax is related to the compatiblity between it and any **location space** (this is the *space that describes the concept of location*, for example it can be the GPS-Location Space where all possible locations are described by GPS coordinates) adopted in a system that use this module because thanks to the separation between *naming* and *locator* system modules, it's possible to make all these characteristics less complex and completly independent.

In order to achieve these goals (modules indepentent between their and the separation of concerns) all parts of the complete studied module, the **Naming System Module** has to maintain all **UIDs** separated from their locations and viceversa the **Locator System Module** has to do the reverse function. Another advantage derived from this **Naming System Module**, as previously described, the module give the **UID** to the consumer entity so, this entity has only to make and send the naming request to the specific module, all details about requests will be explained in the following section.

4.1 Syntax Definition

From the introduction in the previous section, the following **Universal Naming Syntax** can be adopted for any middleware by extending and specializing some characteristics and primitives (for example the entities that will be communicate with this module). In this case it will be explained relating to the TuCSoN middleware as external consumer because it is the particular coordinated middleware considered in this module.
So, this syntax is :

[***entity UID***]

Where this value is the **Universal IDentifier** given by the **Naming System Module** (all details in the previous chapter at the second section) through specified primitives.

In particular, the choosen consumer middleware is TuCSoN so this module is *Java based* (because TuCSoN is *Java based*) but, extending some properties and entities it's possible to adapt this syntax with any middleware because the only requirement is to have a method that allows to obtain an **unique UID** like the TuCSoN primitive that returns an **unique UID** related to the envolved entity; in fact, the **Naming System Module** calls the specific TuCSoN primitive in order to obtain the **UID** and returns this value to the consumer entity.

Now we have to introduce the concept about **naming properties** that is related to the properties that a consumer entity can have. These properties are linked with the entity description in terms of specific requirements (for example a property can be the *logical name* of a consumer entity, used by the developers in order to have a *human universal identifier* like for example

"AGENT PLUTONE"). However all these choices depend on the particular use case about the consumer middleware so, they will be introduce in the development because in this phase the correct execution of this module is guaranteed by the only introduced **UID**, that is the basic requirement.

4.2 Naming System Syntax Procedure

Starting from the previous section, in this one it will be described the sequence of steps about the naming request procedure that a consumer entity has to do. These steps that a consumer entity (relating to the TuCSoN middleware as consumer) has to do in order to obtain an **UID**, through the described module and adopting this syntax, are:

- the consumer entity has to make a naming request specifing the **entity type** (eg: "TuCSoN Agent")
- the consumer sends the created request to the **Outside Interfacing Layer** (contained on the same device)
- wait for the confirm about the request received from the **Outside Interfacing Layer**; in case the request is not valid, it has to restart this procedure
- if the sent request is valid, the answer from the module will be the **UID** because, when the **Outside Interfacing Layer** receive a **naming request**, the specific servant entity of the module has to obtain the **UID** through specific primitives (*Java based* because the TuCSoN environment is *Java*), the **UID** is also stored by the **Internal Request Computation Layer**
- the received **UID** can be associated to the entity

As described in the previous paragraph, this syntax can be used for the TuCSoN middleware as consumer but it can be used with any middleware as consumer but, as previously described, the consumer middleware environment has to give a primitive (or a specific produre) that allows to get an **unique UID** (for example a primitive that return an **UID** given by combination of MAC Address values).

All these characteristics are related to the goal of maintaining the complete module malleability and the orthogonality between the module and the consumer middleware in order to maintain this module completly independent from the consumer.

Chapter 5

Case Studies

This chapter explains all details about the *interaction model* of the **Naming, Locator & Research Module**, with reference to the differences between the *local* and the *distributed* scenario of the module.

The following picture shows the main scenario about the default situation of this module while it's working on many devices.

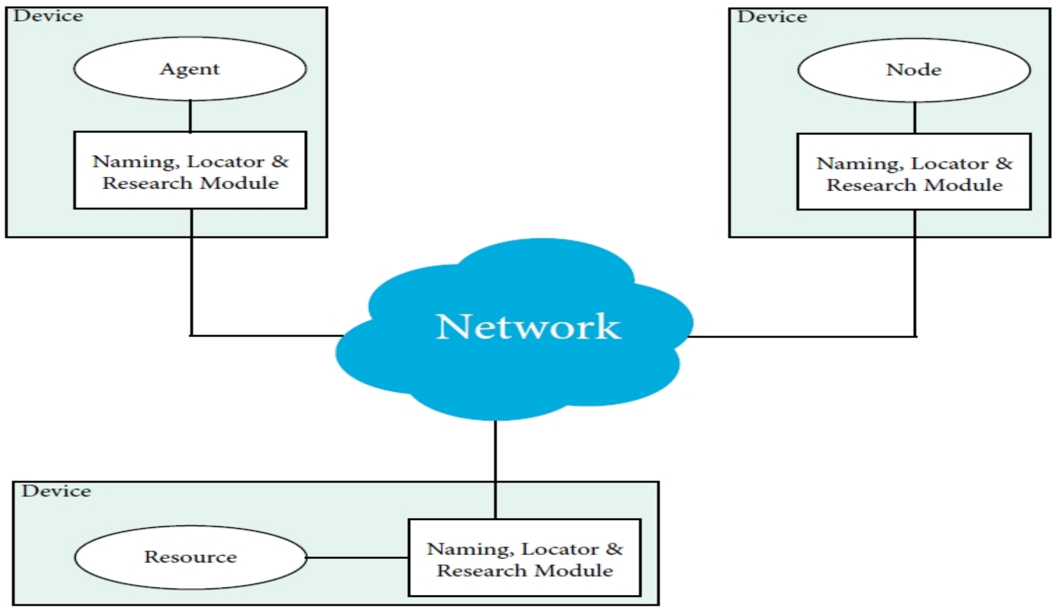

Device
Agent
Naming, Locator & Research Module
Device
Node
Naming, Locator & Research Module
Network
Device
Resource
Naming, Locator & Research Module

Figure 5.1: Main Case Study Interaction Global Model

As you can see in the above picture, this module must be interfaced with the entity who wants to use it in order to request its available services: **Naming Service**, **Locator Service** and **Research Service**.

There are also requirements that the entities involved in a system which use this module has to satisfy, in particular these are refered to the *network-oriented* structure of that system in the case of the network distribution of involved entities; in fact, all entities that want to use this module must be connected to the network and, logically, to the module.

Another important requirement that must be respected is the requirement relating to the *device environment* because, as explained in the **Naming System Syntax** chapter, this module is **Java Based** so, in order to be used from an entity, this entity must running on a device where is available the *Java Virtual Machine* and the **Java Environment**.

Referencing to JADE , you can see that the main structure and the main interction model are very similar to the JADE one; in fact, the main differences between this model and the JADE 's model are related to the architecture in terms of internal entities. The most important is that while in JADE there is a main container for the entities, in this model there is no container because there was the need to have a module that could be used by any entity, without the constrain about the entity location like JADE , where a consumer entity must be placed in a container.

The main model shown in the previous page picture does not show all details about the possible network distribution of this mlodule but it shows only the main situation of this module's use.

All details about internal interactions between components (*internal components* of this module) **distributed** or **local** will be explained in the following sections.

Furthermore the interaction model about all considered use cases will be divided in many parts, depending on the target service's interaction model to describe because this module is too extended in its structure. Also the use cases about the **fault tolerance** will be explained in order to give a complete overview about this module's functioning both in the best working situation as in the failure situation.

5.1 Naming Service System Case Study

In this section the **naming service system**'s interactions model will be explained in order to give a complete overview about this module's part functioning. Particularly, all differences about the *local* and the *distributed* scenario will be described in terms of components interaction and communication.

The following picture shows the internal main interaction between the entities and the **naming service** part of the module, in fact, the white colored components in the picture represent the active components involved in the interaction while the grey colored components represent the unused components.

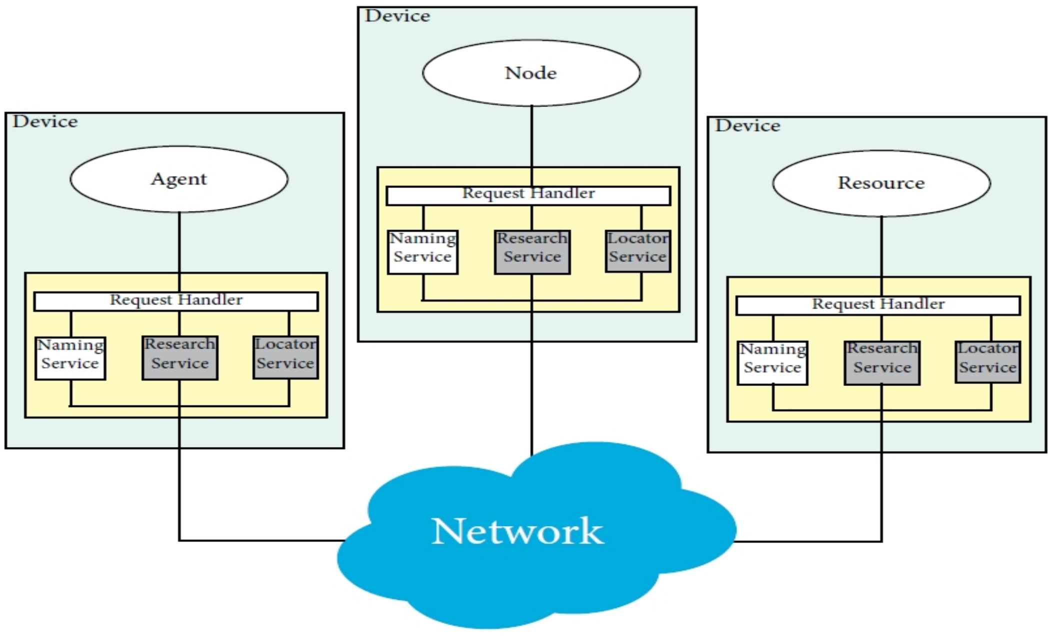


Figure 5.2: Naming Service System Case Study

As you can see in the above picture, the network has a crucial role in the system working cycle because this module works only if the network communication is available (in the local case the network rule is given by the local connection) so, this aspect about the module will be explained in all its details in the following sections.

5.1.1 Local Naming Service System Case Study

In this section all details about the interactions between the **local naming service** (so the module is completely contained on the consumer entity) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **naming service** and an **agent** is shown in the following picture and after it there is the complete description about all interactions.

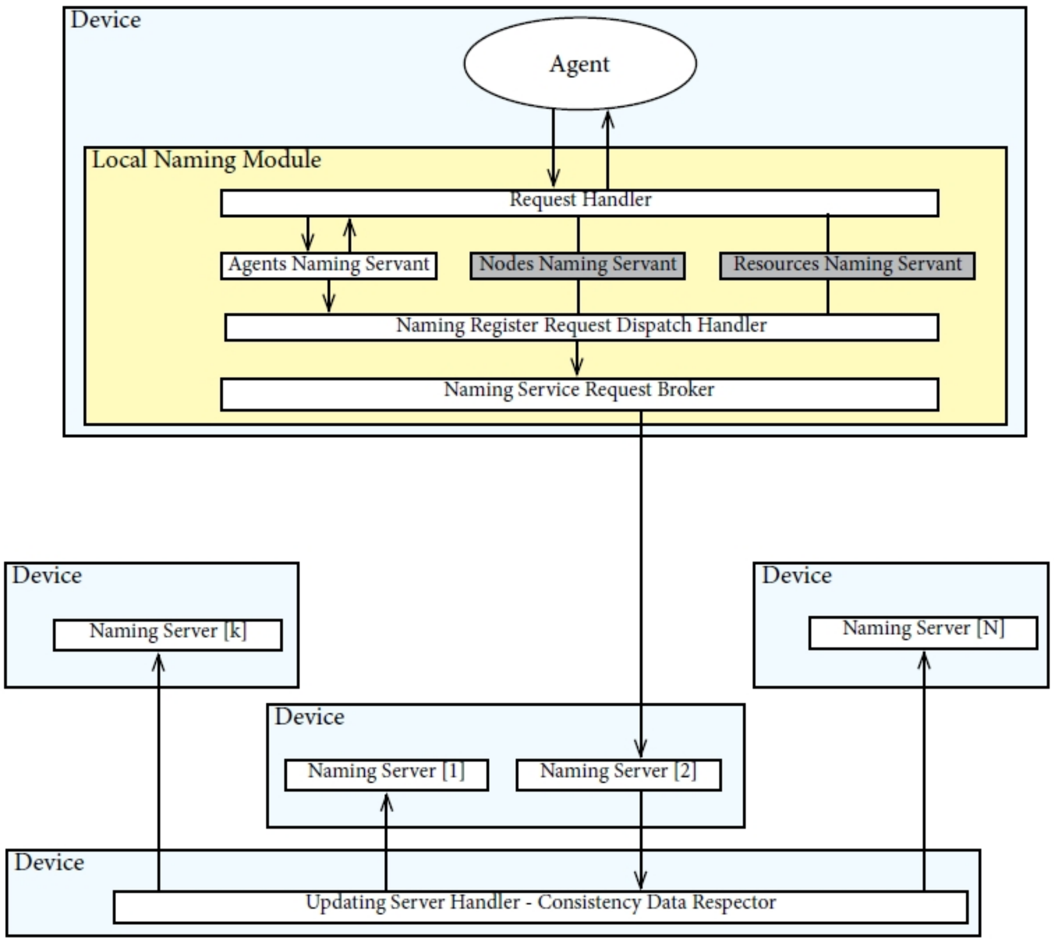


Figure 5.3: Local Naming Service System Case Study

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer agent** makes a *naming request* in order to obtain an **UID** (or with another goal, for example a request in order to set one or many properties about itsel such as its current *state*, etc) specifing its type (**agent**, **node** or **resource**)
- the **consumer agent** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer agent** waits for an answer
- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is an **agent** so the request will be forwarded to the **Agents Naming Servant** entity)
- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier
- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**
- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it
- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming**

Server finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

After the interaction between an **agent** and the naming service, this paragraph contains the interaction description model between the **naming service** and a **node**, that is shown in the following picture and after it there is the complete description about all interactions.

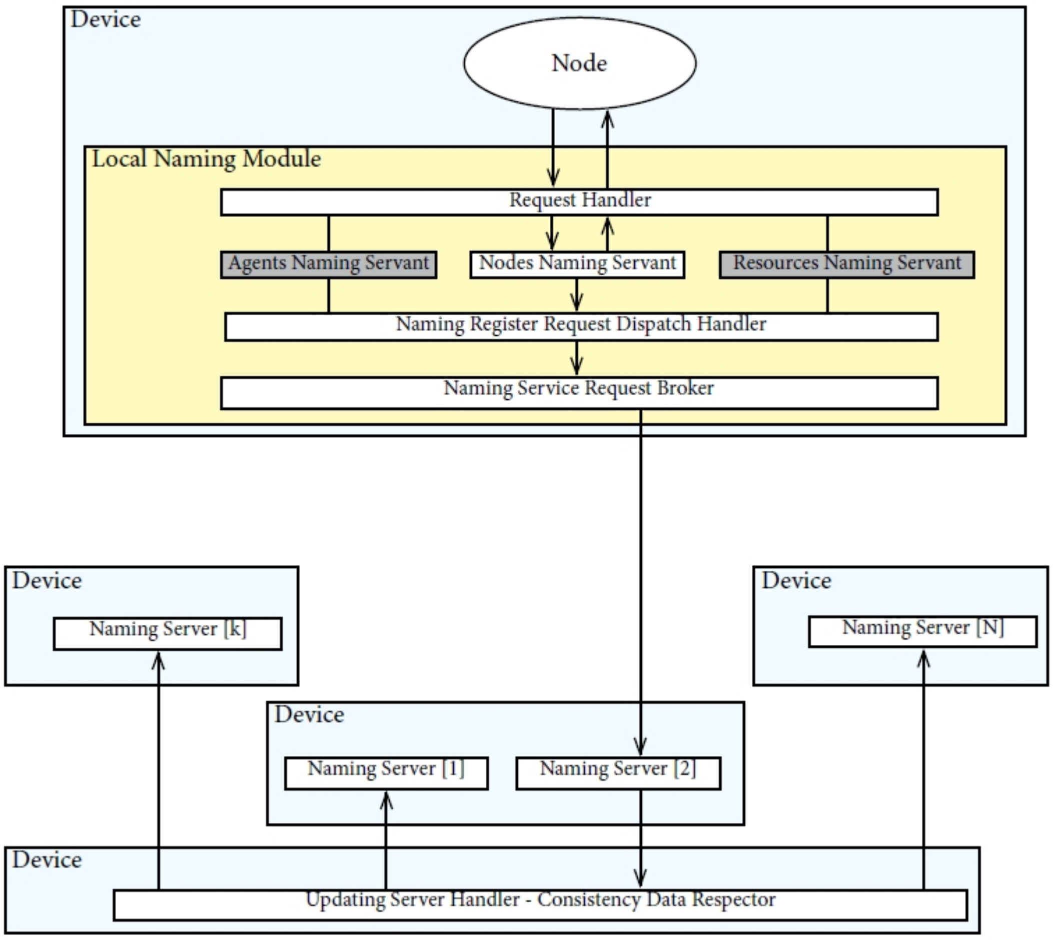


Figure 5.4: Local Naming Node Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer node** makes a *naming request* in order to obtain an **UID** (like the previous described case about the consumer agent) specifing its type (in this case it is **node**)

- the **consumer node** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer node** waits for an answer

- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is a **node** so the request will be forwarded to the **Nodes Naming Servant** entity)

- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity

- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier

- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**

- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming Server** finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated (as explained in the previous case about consumer agents).

The last interaction description to do in the *local naming case* is the interaction description model between the **naming service** and a **resource**, that is shown in the following picture and after it there is the complete description about all interactions.

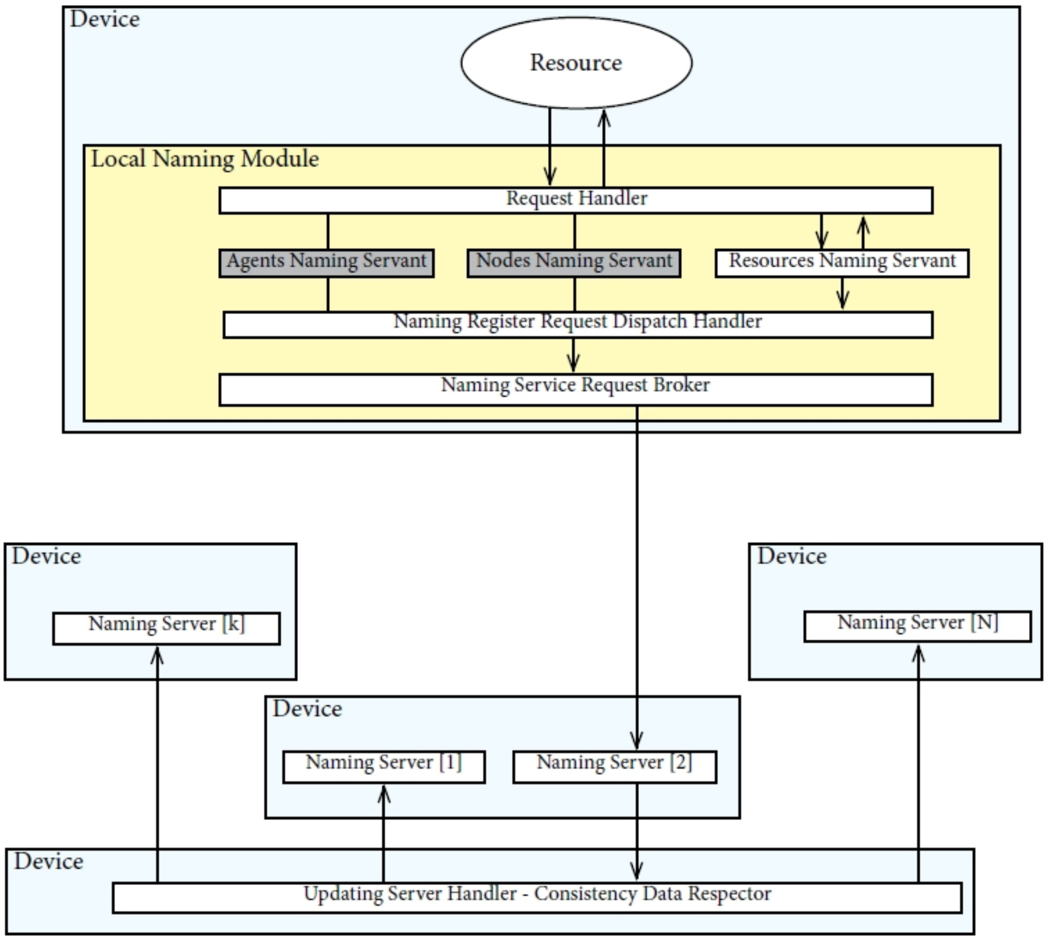


Figure 5.5: Local Naming Resource Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer resource** makes a *naming request* in order to obtain an **UID** (like the previous described cases about the consumer agent and the consumer node) specifing its type (in this case it is **node**)
- the **consumer resource** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer resource** waits for an answer
- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is a **resource** so the request will be forwarded to the **Resource Naming Servant** entity)
- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier
- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**
- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming Server** finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated (as explained in the previous cases about consumer agents and nodes).

5.1.2 Distributed Naming Service System Case Study

In this section all descriptions about the interactions between the **distributed naming service** (so the module is partially contained on the consumer entity and its internal part is located on another, network-reachable device) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **naming service** and an **agent** is shown in the following picture and after it there is the complete description about all interactions.

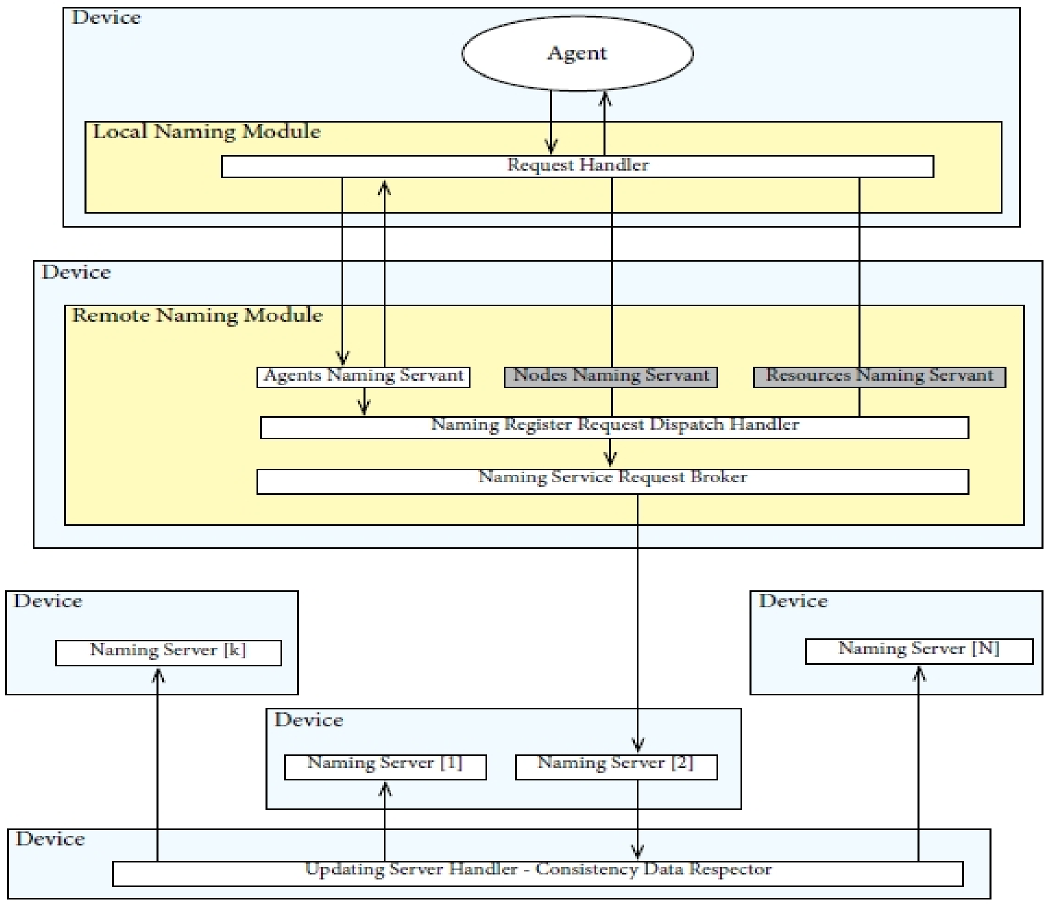


Figure 5.6: Distributed Naming Agent Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer agent** makes a *naming request* in order to obtain an **UID** (or with another goal, for example a request in order to set one or many properties about itsel such as its current *state*, etc) specifing its type (**agent**, **node** or **resource**)
- the **consumer agent** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer agent** waits for an answer
- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is an **agent** so the request will be forwarded to the **Agents Naming Servant** entity). This entity has also to manage the network communication because this is the borderline entity of the consumer entity's device
- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity. This **Naming Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier
- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**

- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it
- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming Server** finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Naming Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

After the interaction between an **agent** and the naming service, this paragraph contains the interaction description model between the **naming service** and a **node**, that is shown in the following picture and after it there is the complete description about all interactions.

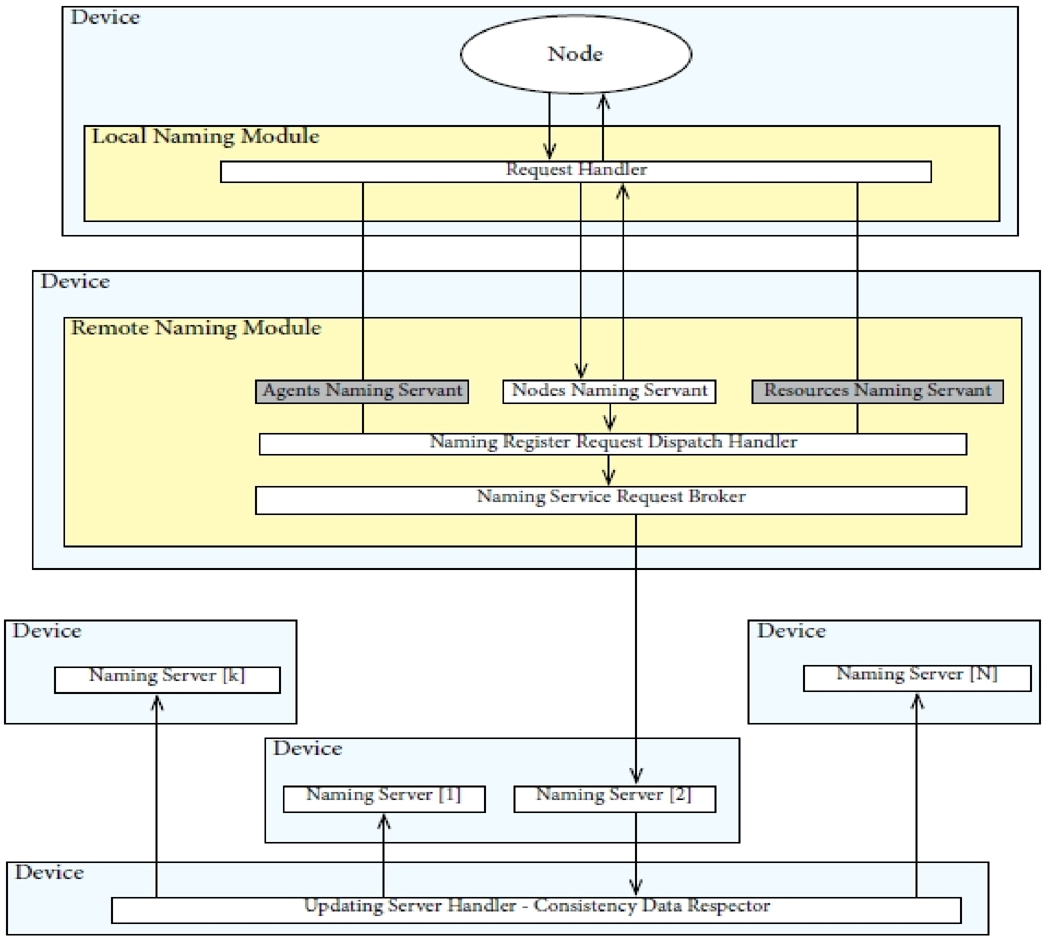


Figure 5.7: Distributed Naming Node Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer node** makes a *naming request* in order to obtain an **UID** (or with another goal, for example a request in order to set one or many properties about itsel such as its current *state*, etc) specifing its type (in this case it is **node**)
- the **consumer node** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer node** waits for an answer
- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is a **node** so the request will be forwarded to the **Nodes Naming Servant** entity). This entity has also to manage the network communication because this is the borderline entity of the consumer entity's device
- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity. This **Naming Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier
- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**

- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming Server** finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Naming Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

The last interaction description to do in the *distributed naming case* is the interaction description model between the **naming service** and a **resource**, that is shown in the following picture and after it there is the complete description about all interactions.

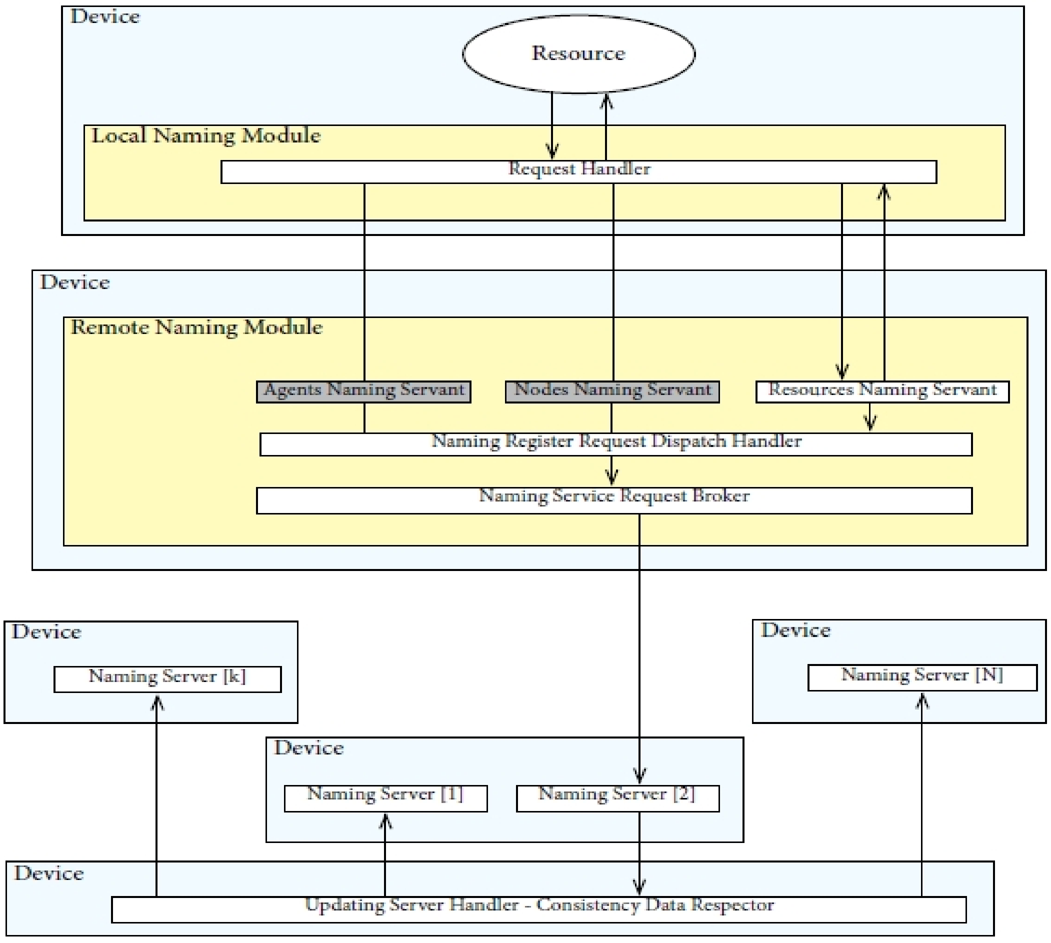


Figure 5.8: Distributed Naming Resource Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **naming service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer resource** makes a *naming request* in order to obtain an **UID** (or with another goal, for example a request in order to set one or many properties about itsel such as its current *state*, etc) specifing its type (in this case it is **resource**)

- the **consumer resource** sends the created *naming request* to the entity able to accept it: the **Request Handler**; so the **consumer resource** waits for an answer

- the **Request Handler** forwards the received request to the specific **Naming Servant** entity based on the consumer entity type (in this case the consumer entity is a **resource** so the request will be forwarded to the **Resources Naming Servant** entity). This entity has also to manage the network communication because this is the borderline entity of the consumer entity's device

- the **Naming Servant** entity builds the **UID** for the consumer entity and makes a new request, so sends it to the **Request Handler** entity in order to give the **UID** to the consumer entity. This **Naming Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device

- while the **Request Handler** entity sends the request answer to the consumer entity, the **Naming Servant** entity sends the created **UID** and makes a new request to the **Naming Register Request Dispatch Handler** entity in order to store the previously created identifier

- the **Naming Register Request Dispatch Handler** accepts the received **UID** and makes a new request in order to forward it to the **broker entity**

- the **Naming Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Naming Server** who answered to the *broker's search request* store the received data (the created **UID**); when the **Naming Server** finish to store the received **UID**, it sends a new request (that contains the **UID**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Naming Servers**, so this entity has the aim to maintain updated all **naming data** in each **Naming Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Naming Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Naming Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Naming Service Broker** entity and the **Naming Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

5.2 Locator Service System Case Study

In this section the **locator service system**'s interaction model will be explained in order to give a complete overview about this module's part functioning.

Particularly, all differences about the *local* and the *distributed* scenario will be described in terms of components interaction and communication.

The following picture shows the internal main interaction between the entities and the **locator service** part of the module, in fact, the white colored components in the picture represent the active components involved in the interaction while the grey colored components represent the unused components.

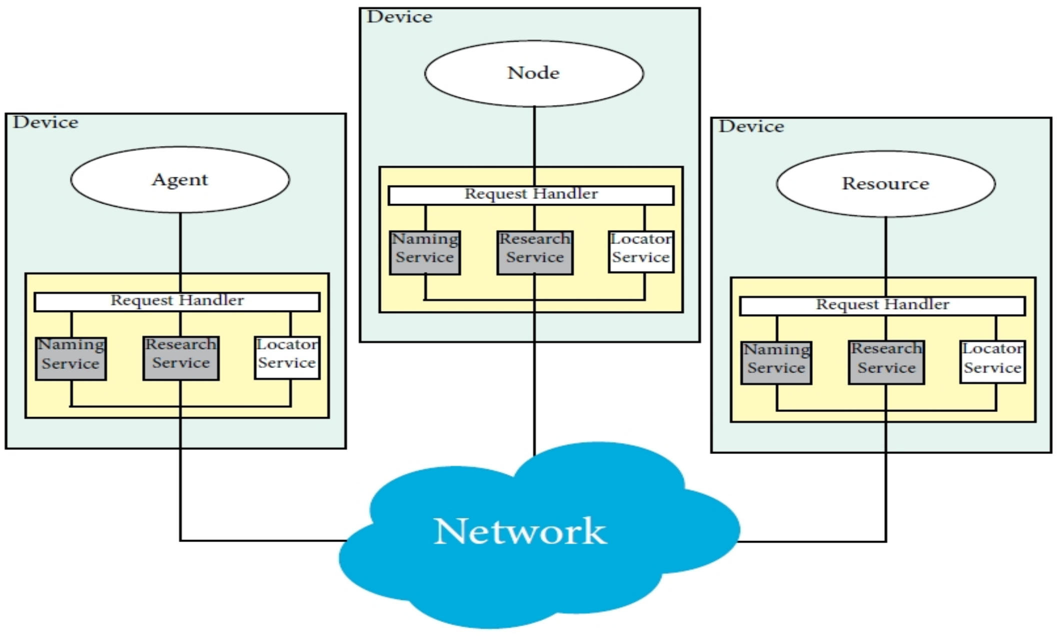


Figure 5.9: Locator Service System Case Study

As you can see in the above picture, the network has a crucial role in the system working cycle because this module works only if the network communication is available (in the local case the network rule is given by the local connection) so, this aspect about the module will be explained in all its details in the following sections.

5.2.1 Local Locator Service System Case Study

In this section all details about the interactions between the **local locator service** (so the module is completely contained on the consumer entity) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **locator service** and an **agent** is shown in the following picture and after it there is the complete description about all interactions.

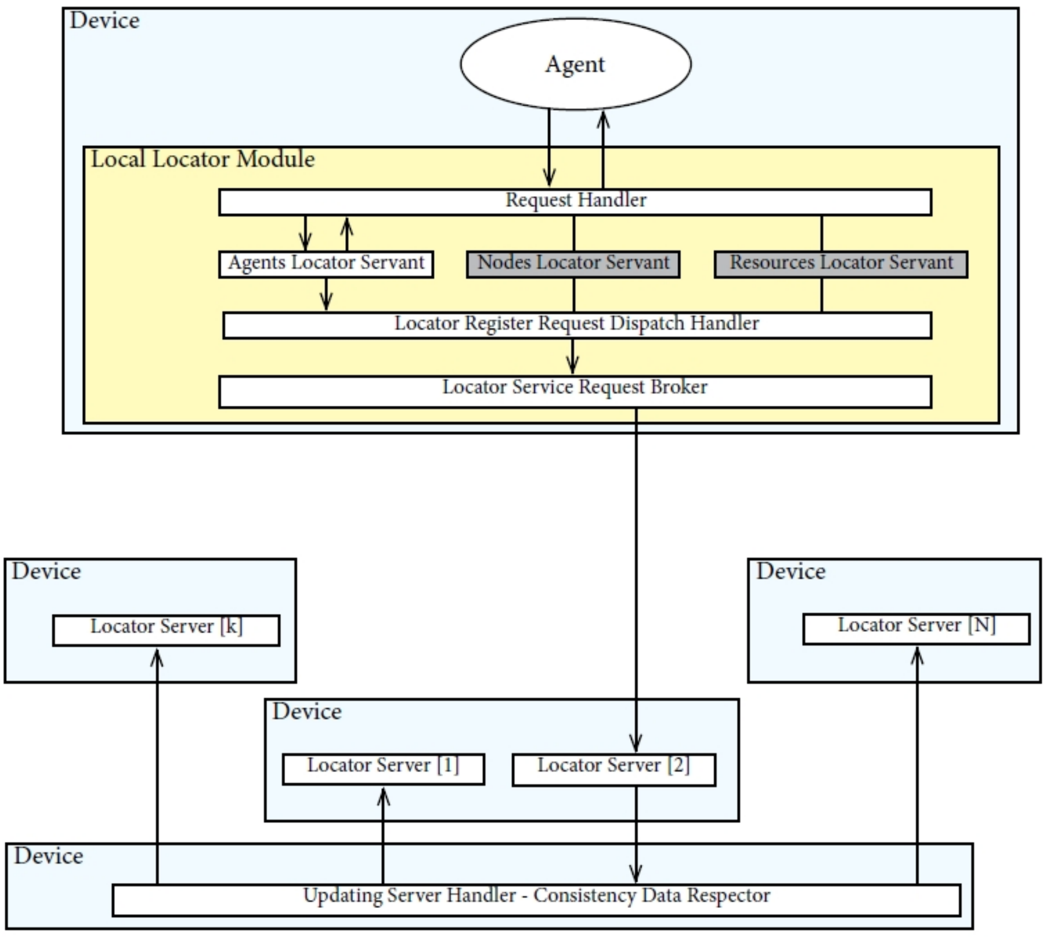


Figure 5.10: Local Locator Agent Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer agent** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (**agent**, **node** or **resource**)

- the **consumer agent** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer agent** waits for an answer

- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is an **agent** so the request will be forwarded to the **Agents Locator Servant** entity)

- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity

- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data

- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**

- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

After the interaction between an **agent** and the locator service, this paragraph contains the interaction description model between the **locator service** and a **node**, that is shown in the following picture and after it there is the complete description about all interactions.

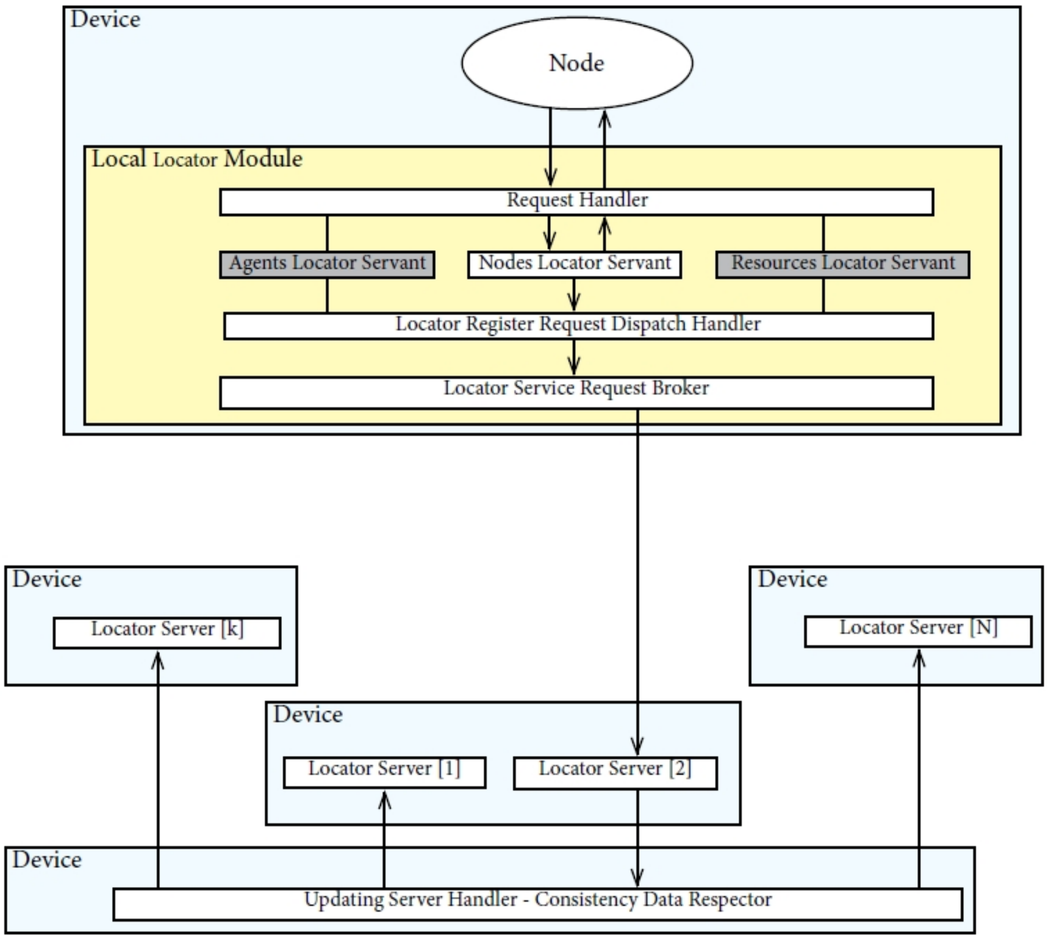


Figure 5.11: Local Locator Node Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer node** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (in this case it is **node**)
- the **consumer node** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer node** waits for an answer
- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is a **node** so the request will be forwarded to the **Nodes Locator Servant** entity)
- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data
- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**
- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

The last interaction description to do in the *local locator case* is the interaction description model between the **locator service** and a **resource**, that is shown in the following picture and after it there is the complete description about all interactions.

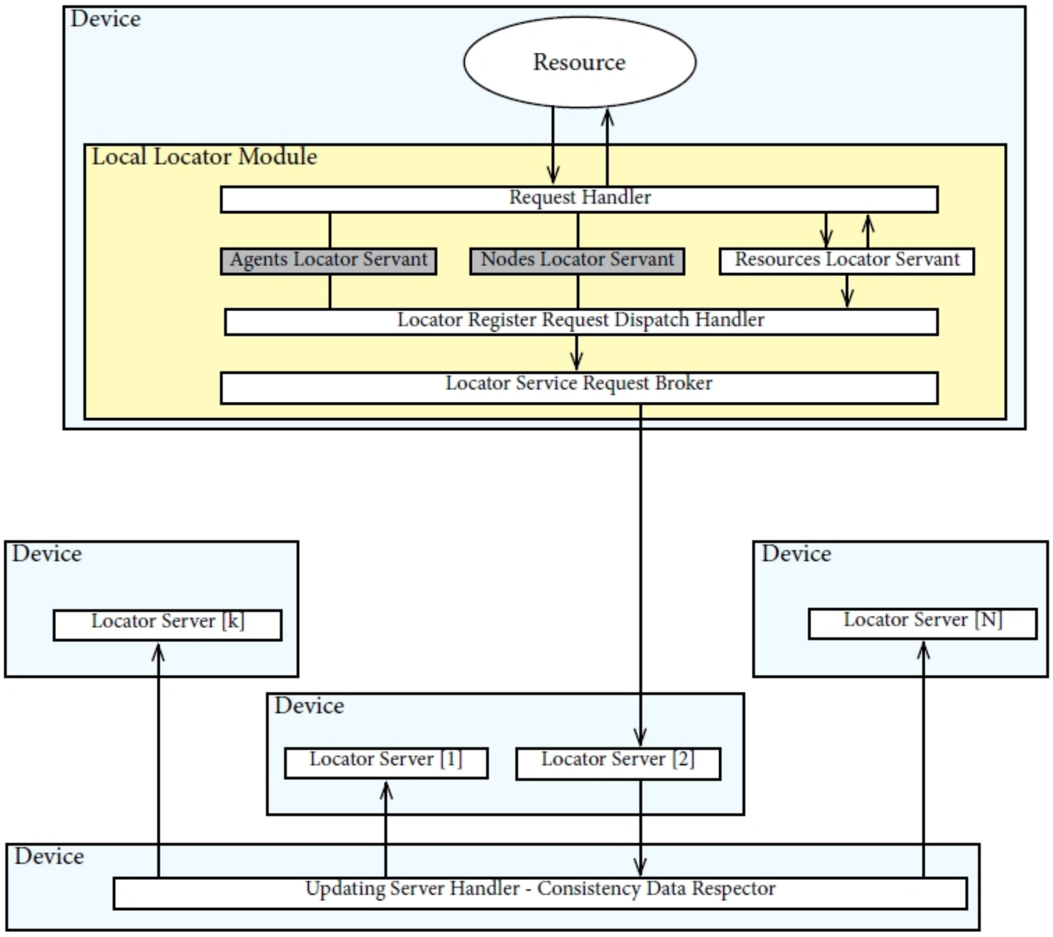


Figure 5.12: Local Locator Resource Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer resource** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (in this case it is **Resource**)
- the **consumer resource** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer resource** waits for an answer
- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is a **resource** so the request will be forwarded to the **Resources Locator Servant** entity)
- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data
- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**
- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

5.2.2 Distributed Locator Service System Case Study

In this section all descriptions about the interactions between the **distributed locator service** (so the module is partially contained on the consumer entity and its internal part is located on another, network-reachable device) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **locator service** and an **agent** is shown in the following picture and after it there is the complete description about all interactions.

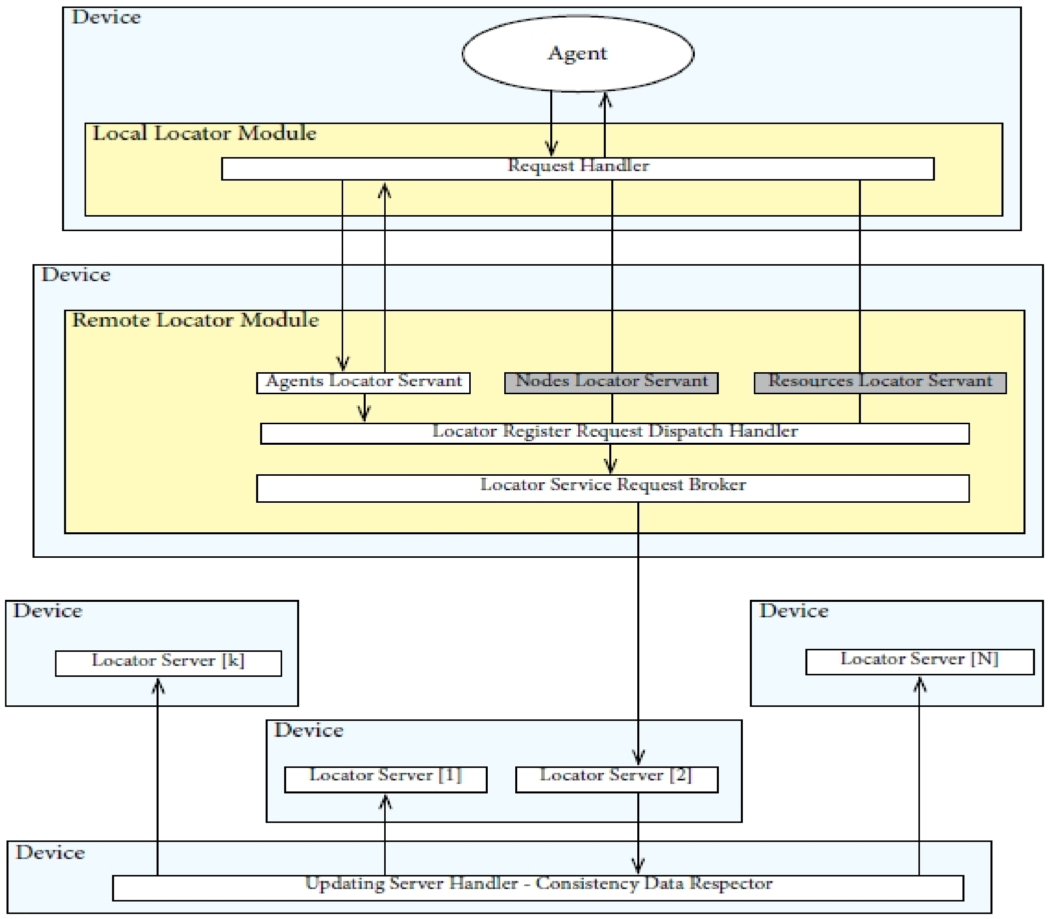


Figure 5.13: Distributed Locator Agent Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer agent** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (**agent**, **node** or **resource**)
- the **consumer agent** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer agent** waits for an answer
- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is an **agent** so the request will be forwarded to the **Agents Locator Servant** entity). This entity has also to manage the network communication because this is the borderline entitiy of the consumer entity's device
- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity. This **Locator Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data
- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**

- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it
- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Locator Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

After the interaction between an **agent** and the locator service, this paragraph contains the interaction description model between the **locator service** and a **node**, that is shown in the following picture and after it there is the complete description about all interactions.

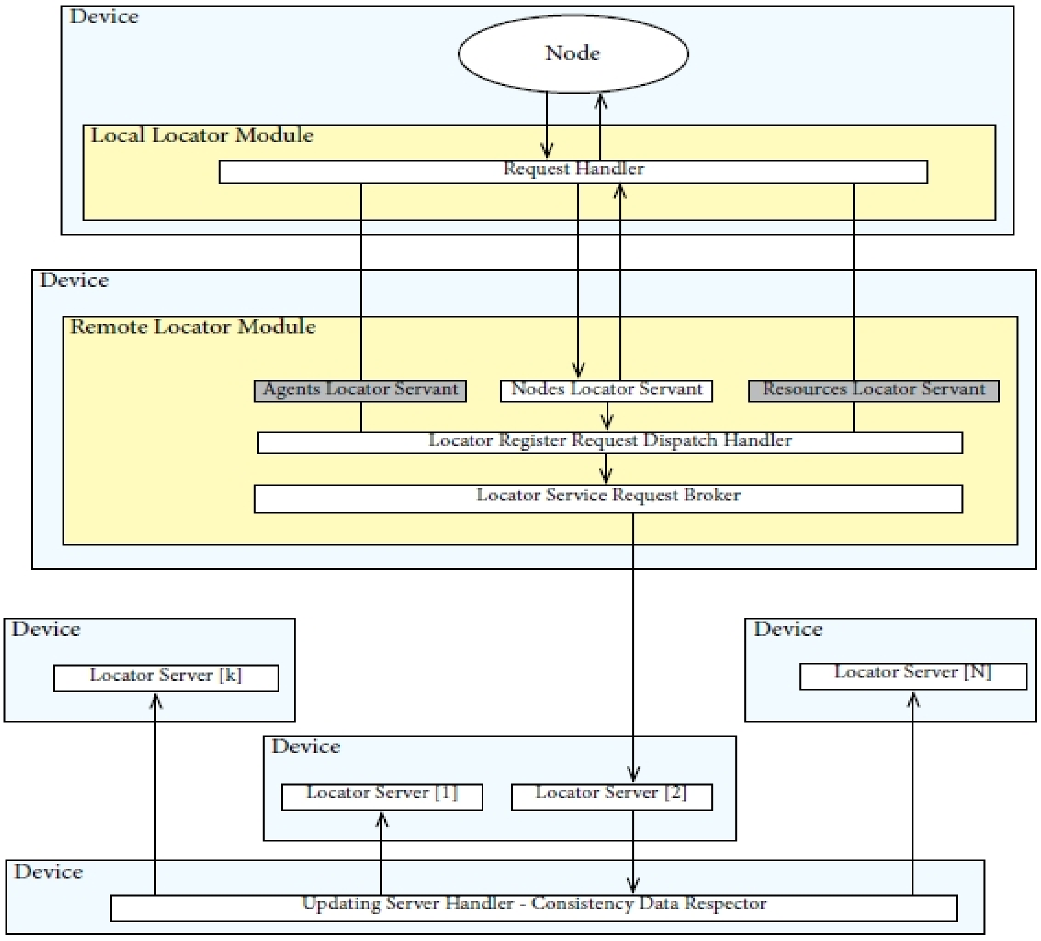


Figure 5.14: Distributed Locator Node Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer node** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (in this case it is **node**).
- the **consumer node** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer node** waits for an answer
- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is a **node** so the request will be forwarded to the **Nodes Locator Servant** entity). This entity has also to manage the network communication because this is the borderline entitiy of the consumer entity's device
- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity. This **Locator Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data
- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**

- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Locator Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

The last interaction description to do in the *distributed locator case* is the interaction description model between the **locator service** and a **resource**, that is shown in the following picture and after it there is the complete description about all interactions.

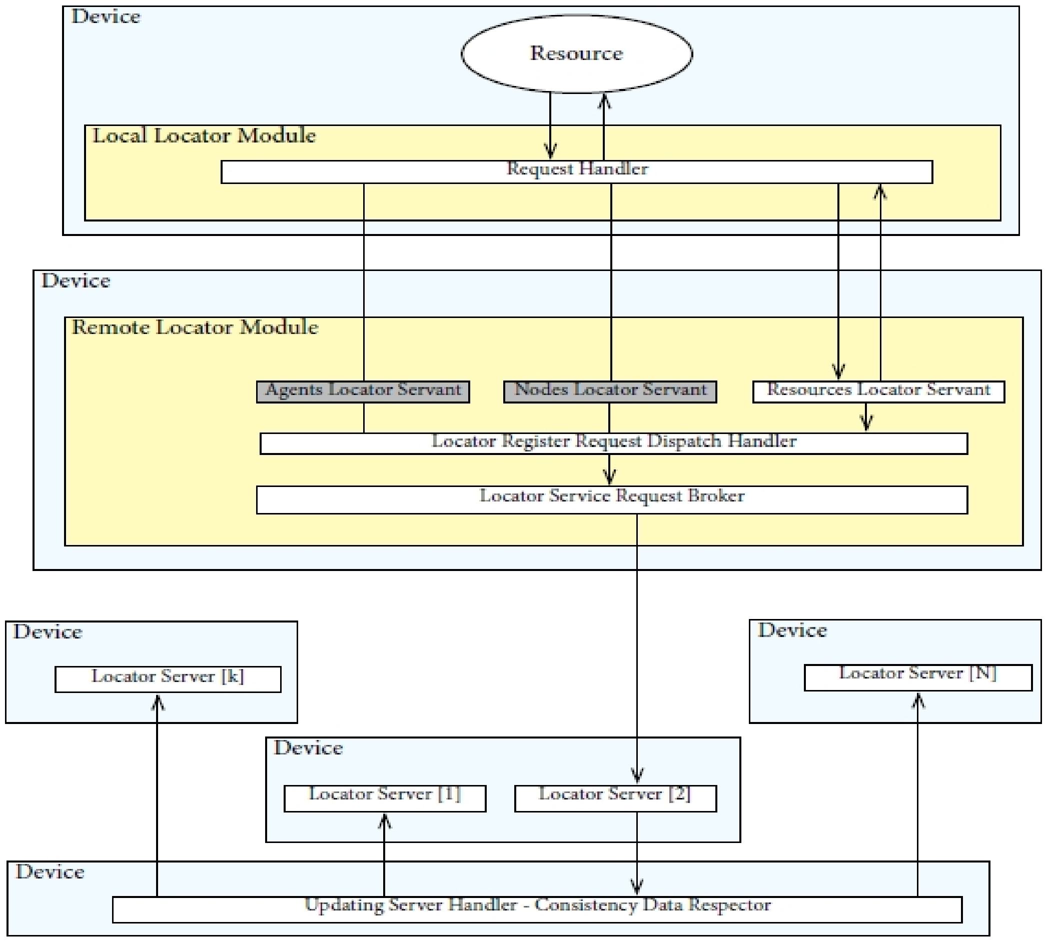


Figure 5.15: Distributed Locator Resource Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **locator service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer resource** makes a *locator request* in order to register its current **location** (or with another goal depending on the location concept, for example a request in order to set its current *GPS* location, etc) specifing its type (in this case it is **resource**).

- the **consumer resource** sends the created *locator request* to the entity able to accept it: the **Request Handler**; so the **consumer resource** waits for an answer

- the **Request Handler** forwards the received request to the specific **Locator Servant** entity based on the consumer entity type (in this case the consumer entity is a **node** so the request will be forwarded to the **Resources Locator Servant** entity). This entity has also to manage the network communication because this is the borderline entitiy of the consumer entity's device

- the **Locator Servant** entity accept the received request and create the request to send to the consumer entity; so the **Locator Servant** sends it to the **Request Handler** entity in order to give the answer to the consumer entity. This **Locator Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device

- while the **Request Handler** entity sends the request answer to the consumer entity, the **Locator Servant** entity creates a new request with all the location data to register and this servant sends it to the **Locator Register Request Dispatch Handler** entity in order to store the received location data

- the **Locator Register Request Dispatch Handler** accept the received request with all location data and makes a new request in order to forward it to the **broker entity**

- the **Locator Service Request Broker** search the current free server in order to serve the received request from the upper entity; when a server if available (so when a server answer to the *free server search request* sent), this **broker entity** sends data to it

- the free **Locator Server** who answered to the *broker's search request* store the received data (all the **location data**); when the **Locator Server** finish to store the received data, it sends a new request (that contains the **registered data**) to the **Updating Server Handler** entity (also called **Consistency Respector**) in order to update this value in all **Locator Servers**, so this entity has the aim to maintain updated all **locator data** in each **Locator Server** thus this entity is a **Consistency Respector** (this is the motivation for this entity's second name)

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Locator Servers** are *distributed*, like the **Updating Server Handler**, the **Request Handler** and the module core entities.

The management of the communication between the **Request Handler** and the **Locator Servant** is aim of these involved borderline entities as described in the above working steps.

The management of the communication between the **Locator Service Broker** entity and the **Locator Servers** is aim of the involved entities because they are the borderline entities of their parts of the module.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

5.3 Research Service System Case Study

In this section the **research service system**'s interaction model will be explained in order to give a complete overview about this module's part functioning.

Particularly, all differences about the *local* and the *distributed* scenario will be described in terms of components interaction and communication.

The following picture shows the internal main interaction between the entities and the **research service** part of the module, in fact, the white colored components in the picture represent the active components involved in the interaction while the grey colored components represent the unused components.

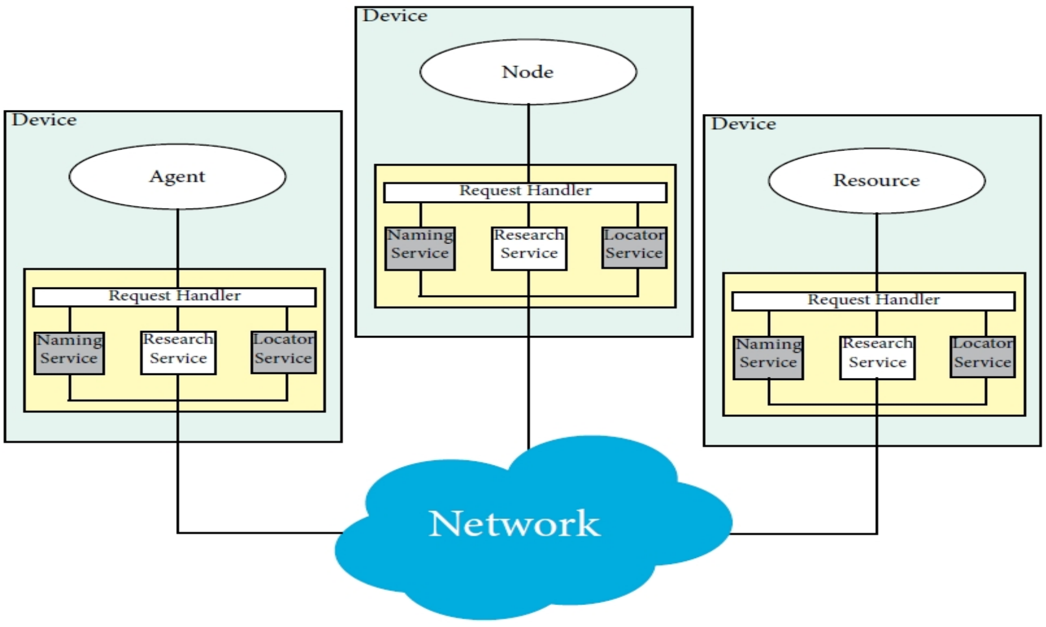


Figure 5.16: Research Service System Case Study

As you can see in the above picture, the network has a crucial role in the system working cycle because this module works only if the network communication is available (in the local case the network rule is given by the local connection) so, this aspect about the module will be explained in all its details in the following paragraphs.

5.3.1 Local Research Service System Case Study

In this section all details about the interactions between the **local research service** (so the module is completely contained on the consumer entity) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **research service** and a **consumer entity** (there are no differences deriving from the consumer entity type so the schema contains the *generic consumer entity*) is shown in the following picture and after it there is the complete description about all interactions.

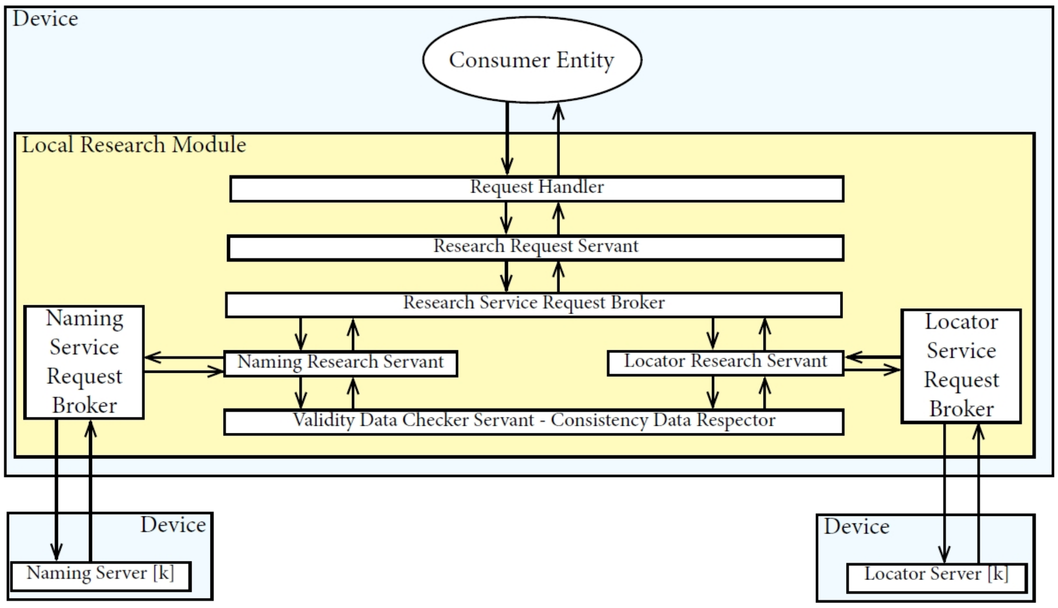


Figure 5.17: Local Research Main Case Study Interaction Model

As you can see in the above picture there are many interactions between the consumer entity and the **research service** and many interactions between the internal entities.

In details, the working steps are the following.

- the **consumer entity** makes a *research request* in order to obtain informations about another entity about its naming or location data, the **consumer entity** has also to specify its type (**agent**, **node** or **resource**).
- the **consumer entity** sends the created *research request* to the entity able to accept it: the **Request Handler**; so the **consumer entity** waits for an answer
- the **Request Handler** forwards the received request to the **Research Request Servant** entity).
- the **Research Request Servant** entity accepts the received request, so it creates a particular answer to send to the consumer entity in order to communicate to it that the request has been accepted (when the consumer entity will receive this confirm answer, it will wait the request data answer); in the end the **Research Request Servant** sends the confirm answer to the **Request Handler** entity in order to give it to the consumer entity.
- while the **Request Handler** entity sends the request answer to the consumer entity, the **Research Request Servant** entity creates a new request with all data about the received request and sends it to the **Research Service Request Broker** entity in order to satisfy the received request
- the **Research Service Request Broker** creates two requests:
  - a request that will be sent to the **Naming Research Servant** in order to check the naming informations about the involved entities in the received request from the **consumer entity**
  - a request that will be sent to the **Locator Research Servant** in order to check the location informations about the involved entities in the received request from the **consumer entity**
- the **Naming Research Servant** receive the request from the **Research Service Request Broker** and creates a new request to send

to the **Naming Service Request Broker** that contains all data to check the informations of the involved entities. The **Naming Service Request Broker** answer to the **Naming Research Servant**'s request by interactions with the current free **Naming Server**. In the end the **Naming Service Request Broker** sends all data to the **Naming Research Servant**

- the **Locator Research Servant** receive the request from the **Research Service Request Broker** and creates a new request to send to the **Locator Service Request Broker** that contains all data to check the informations of the involved entities. The **Locator Service Request Broker** answer to the **Locator Research Servant**'s request by interactions with the current free **Locator Server**. In the end the **Locator Service Request Broker** sends all data to the **Locator Research Servant**

- the **Naming Research Servant** and the **Locator Research Servant** send the received data from each **Service Request Broker** to the **Validity Data Checker Servant**; that entity has to check the validity of each received data about the **Naming Informations** and the **Location Informations** and communicates to each **Research Servant** the answer in order to check each information (this entity check the data validity so it checks also the consistency of all informations, then this entity is also called **Consistency Data Respector**)

- each **Research Servant** create the answer that will be sent to the **Research Service Request Broker**; this answer contains all informations and all data in order to create the answer to the **consumer entity**'s **research request**

- the **Research Service Request Broker** creates the answer that will be sent to the **Research Request Servant** in order to give all informations requested in the **consumer entity**'s received **reseach request**

- the **Research Request Servant** sends the received answer from the **Research Service Request Broker** to the **Request Handler** after the checking of all informations contained in the received message
- the **Request Handler** sends the answer to the **consumer entity** in order to satisfy its initial received **research request**

In these working steps are described also network interactions between the **Service Request Broker** entities and its specific **Server entity** that we have not to describe in this section because they are explained in the previously seen sections about the **Naming Service** and the **Locator Service**.

All details about the **fault tolerance** issues due to communication problems (for example the crash of an entity, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

5.3.2 Distributed Research Service System Case Study

In this section all details about the interactions between the **distributed research service** (so the module is completely contained on the consumer entity) and the entities will be explained, dividing the cases based on the consumer entity type (**agent**, **node** and **resource**, in this case it is **agent**).

The interaction description model between the **research service** and a **consumer entity** (there are no differences deriving from the consumer entity type so the schema contains the *generic consumer entity*) is shown in the following picture and after it there is the complete description about all interactions.

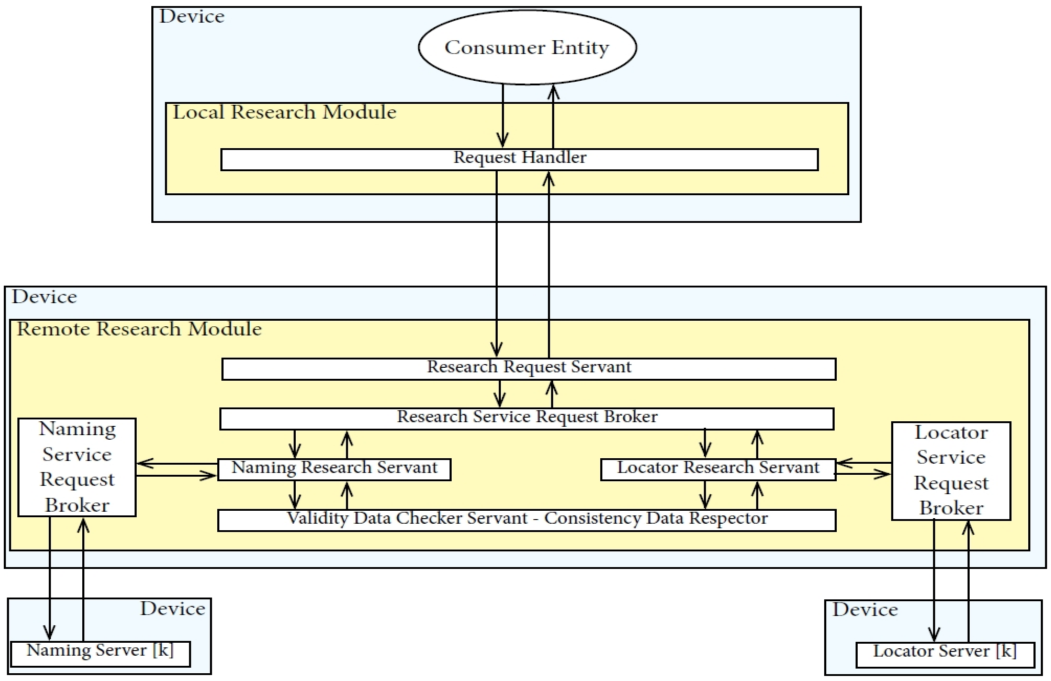


Figure 5.18: Distributed Research Main Case Study Interaction Model

As you can see in the above picture, the network has a crucial role in the system working cycle because this module works only if the network communication is available (in the local case the network rule is given by the local connection) so, this aspect about the module will be explained in all its details in the following paragraphs.

In details, the working steps are the following.

- the **consumer entity** makes a *research request* in order to obtain informations about another entity about its naming or location data, the **consumer entity** has also to specify its type (**agent**, **node** or **resource**).

- the **consumer entity** sends the created *research request* to the entity able to accept it: the **Request Handler**; so the **consumer entity** waits for an answer

- the **Request Handler** forwards the received request to the **Research Request Servant** entity). This entity has also to manage the network communication because this is the borderline entity of the consumer entity's device

- the **Research Request Servant** entity accepts the received request, so it creates a particular answer to send to the consumer entity in order to communicate to it that the request has been accepted (when the consumer entity will receive this confirm answer, it will wait the request data answer); in the end the **Research Request Servant** sends the confirm answer to the **Request Handler** entity in order to give it to the consumer entity. This **Research Request Servant** has also to manage the network communication with the **Request Handler** entity because this is the borderline entity of the module core's device

- while the **Request Handler** entity sends the request answer to the consumer entity, the **Research Request Servant** entity creates a new request with all data about the received request and sends it to the **Research Service Request Broker** entity in order to satisfy the received request

- the **Research Service Request Broker** creates two requests:

  - a request that will be sent to the **Naming Research Servant** in order to check the naming informations about the involved entities in the received request from the **consumer entity**

  - a request that will be sent to the **Locator Research Servant** in order to check the location informations about the involved entities in the received request from the **consumer entity** and that request has also the aim to obtain all location informations to satisfy the received request

- the **Naming Research Servant** receive the request from the **Research Service Request Broker** and creates a new request to send to the **Naming Service Request Broker** that contains all data to check the informations of the involved entities. The **Naming Service Request Broker** answer to the **Naming Research Servant**'s request by interactions with the current free **Naming Server**. In the end the **Naming Service Request Broker** sends all data to the **Naming Research Servant**

- the **Locator Research Servant** receive the request from the **Research Service Request Broker** and creates a new request to send to the **Locator Service Request Broker** that contains all data to check the informations of the involved entities. The **Locator Service Request Broker** answer to the **Locator Research Servant**'s request by interactions with the current free **Locator Server**. In the end the **Locator Service Request Broker** sends all data to the **Locator Research Servant**

- the **Naming Research Servant** and the **Locator Research Servant** send the received data from each **Service Request Broker** to the **Validity Data Checker Servant**; that entity has to check the validity of each received data about the **Naming Informations** and the **Location Informations** and communicates to each **Research Servant** the answer in order to check each information (this entity check the data validity so it checks also the consistency of all informations, then this entity is also called **Consistency Data Respector**)

- each **Research Servant** create the answer that will be sent to the **Research Service Request Broker**; this answer contains all informations and all data in order to create the answer to the **consumer entity**'s **research request**

- the **Research Service Request Broker** creates the answer that will be sent to the **Research Request Servant** in order to give all informations requested in the **consumer entity**'s received **reseach request**

- the **Research Request Servant** sends the received answer from the **Research Service Request Broker** to the **Request Handler** after the checking of all informations contained in the received message

- the **Request Handler** sends the answer to the **consumer entity** in order to satisfy its initial received **research request**

Now we have to introduce all details about the management of the network communication in these described interactions because, as you can see in the above picture, the **Research Service Core** and the **Research Service Interfacing Module** are *distributed*, in fact the only network communication contained in this module is the communication between the **Request Handler** and the **Research Request Servant** because other communications are related to the **Service Request Broker** entities that depend on other modules (the **naming service module** and the **locator service module**).

The management of the communication between the **Request Handler** and the **Research Request Servant** is aim of these involved borderline entities as described in the above working steps.

All details about the **fault tolerance** issues due to communication problems (for example the network connection of a device is temporarily unavailable, etc) or request problems (for example if a received request has an invalid syntax, etc) will be described in the last part of this chapter because it involves all case studies treated.

5.4 Fault Tolerance Case Study

The last case studies section is devoted to *fault tolerance*'s analysis.

The goal of this section is to explain all details about the *fault tolerance* in order to provide guidelines to solve all the problems related to the *reliability* and *responsiveness* of the system in case of failure or malfunction of the module or network.

The **fault tolerance** is the *ability of a system does not suffering failures* (ie intuitively service interruptions) even in the presence of faults. Fault tolerance is one of the aspects that define the *reliability*. It is important to note that the fault-tolerance does not guarantee immunity from all faults, but it only guarantee the manage of faults in order to provides a minimun service level in all situation.

The first main aspect that we have to consider is that in each *fault tolerance* case the presence of these contingencies is signaled to the other modules distributed on the network on the other devices to communicate with other *consumer entities* that there is a critical situation due to an error or a fault; this situation is shown in the following picture.

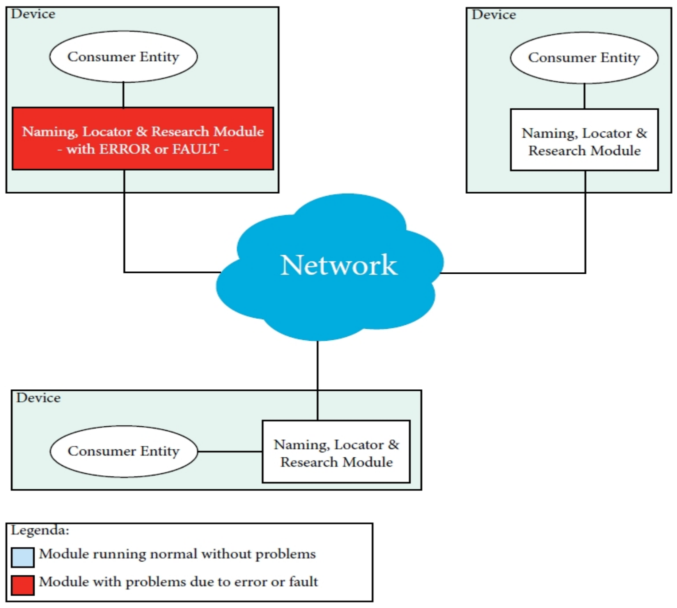

Device
Consumer Entity
Naming, Locator & Research Module - with ERROR or FAULT -
Device
Consumer Entity
Naming, Locator & Research Module
Network
Device
Consumer Entity
Naming, Locator & Research Module
Legenda:
Module running normal without problems
Module with problems due to error or fault

Figure 5.19: Fault Tolerance Main Case Study Interaction Model

After the *fault tolerance*'s introduction, we have to describe all case studies with all details in order to give a detailed description of all issues.

In this section only the main general **fault tolerance case studies** will be described; these case studies are:

- **Internal Entities Crash**: this situation is related to the crash of an internal entity of the module
- **Unavailable Connection between Internal Entities**: this situation is related to the unavailability of the connection between the internal entities of the module
- **Unavailable Network Connection**: this situation is related to the network connection's unavailability. This connection's unavailability can be:
  - **Temporary**: in this case the network connection is unavailable for a short limited time
  - **Persistent**: in this case the network connection is unavailable for a long time, so the connection can't be restored
- **Server Updating Error**: in this case the updating process of data (based on Cassandra , as seen in the first chapter) generates an error due to a **Database Updating Error**, so in this case the updating of a data on the database generates an error

An important aspect to explain is the concept of **connection** because in this section it is a foundamental concept. In this section with the term of **connection** we mean the general connection, not only the *internet* one because this module can be distributed on the network across *internet* or across a *local network*. So, when we will talk about the **connection**, it is related to these kind of connection.

5.5 Fault Tolerance Case Study: Internal Entities Crash

This section contains all details about the **fault tolerance case study** relating to the **crash of the internal entities** contained in this module.

First of all we need to explain all details about the creation and the lifecycle of the internal entities. An internal entity starts when the module starts and it creates a **child entity** that has the aim to serve the incoming requests.

This paradign has been adopted in order to prevent failures due to entities crash because if an entity crashies, this entity is a **child entity** so, the **father entity** can replace the crashed entity by creation of a new child that will serve the pending reques.

The described main **internal entity's life cycle** will be shown in the following picture.

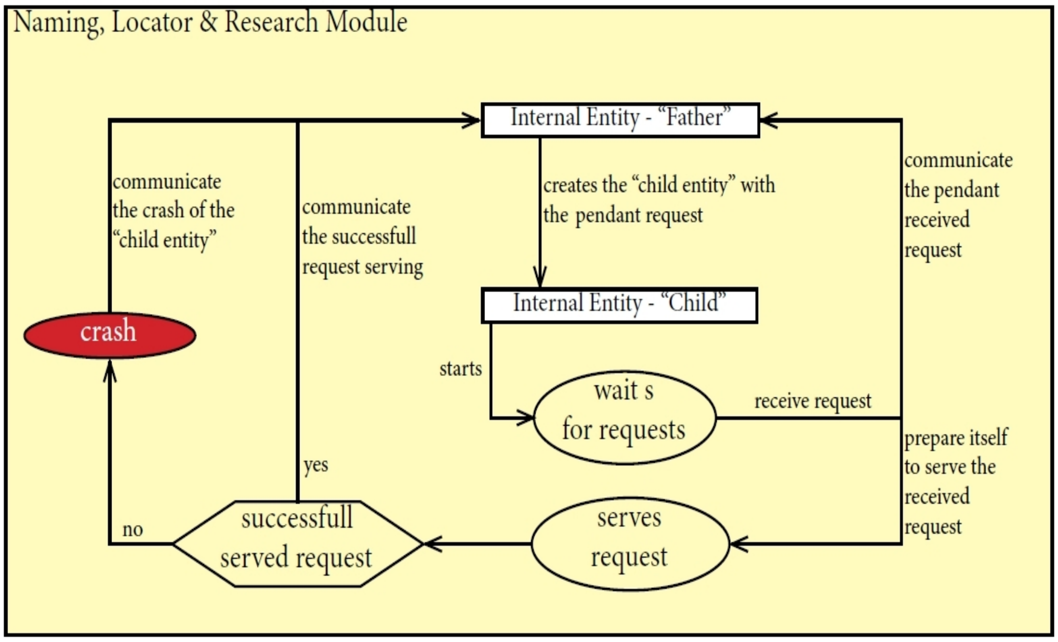

Naming, Locator & Research Module
Internal Entity - "Father"
communicate the crash of the "child entity"
communicate the successfull request serving
creates the "child entity" with the pendant request
communicate the pendant received request
Internal Entity - "Child"
crash
starts
wait s for requests
receive request
prepare itself to serve the received request
yes
no
successfull served request
serves request

Figure 5.20: Fault Tolerance Internal Entities Life-Cycle Crash Case Study Interaction Model

In details, the working steps are the following:

- an **Internal Entity** has been created, this entity is the **"father"** entity of the server entity that will have the aim to serve all requests that will be received. When this entity is created, it starts, so it creates a **"child entity"**. The father entity will pass to the child entity the eventually pending request (this request derives by a previous crash of another child entity)
- **"child entity"** starts and waits for requests. If there is a pending request received from the father entity, this entity will go to the next step (in that step it will serve the request). If there isn't a pending request, it will wait for an incoming request
- when the child entity serves the target request, if there is a fault due to a **crash** of that entity, this event will be communicated to the father entity in order to create a new child entity with the aim to serve requests. Else, if the request has been successfully served, this event will be communicated to the father entity that will delete its pending request (this request is the child entity's served request)
- if the father entity receives a notification of the **child entity crash**, it will create a new child entity that will do the previous steps starting from the second one described

This paradigm is the one used by servers in order to guarantee a better efficiency of the server system (in this case it is represented by the internal entities that have the aim to serve all incoming requests).

Those working steps are related to any **module's internal entity**, so each entity has the role of **father entity** that has to create all needed **child entities** because there can be many **child entities** created by a single **father entity** in order to serve many incoming requests concurrently.

In any case of **Internal Entities Crash**, a notification will be sent to the **Outside Interfacing Layer** (composed by the **Request Handler** entity) in order to manage all problems do to this critical situation. In fact, the **Request Handler** can maintain a cache where all pending requests are inserted while the internal critical situation continues in order to maintain the module's reliability.

5.6 Fault Tolerance Case Study: Unavailable Connection between Internal Entities

This section contains all details about the **fault tolerance case study** relating to the **connection unavailability between internal entities** contained in this module.

This issue is foundamental because if an entity runs normally, it can work only with a connection with others because this is a requirements due to the request and data passing between all involved entities.

First of all we need to explain all details about the initializing and establishment of the connection between the internal entities because this discussion is related to the previously seen life cycle of the father and child entities. When each entity starts (both father entity and child entity) it has to establish a connection with others entities which it has to communicate (for example in order to forward received requests).

This procedure has the aim to guarantee the correct communication between the internal entities in order to guarantee the correct working of the module. The described main procedure will be shown in the following picture.

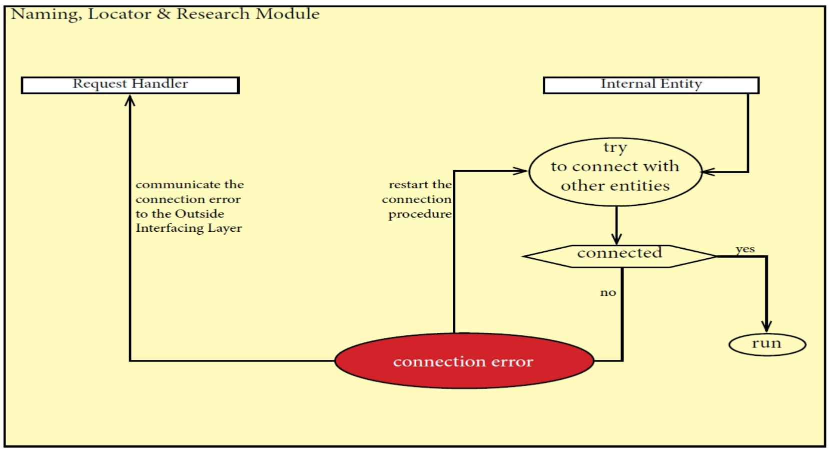

Naming, Locator & Research Module
Request Handler
Internal Entity
try to connect with other entities
communicate the connection error to the Outside Interfacing Layer
restart the connection procedure
connected
yes
no
run
connection error

Figure 5.21: Fault Tolerance Unavailable Connection between Internal Entities Case Study Interaction Model

In details, the working steps are the following:

- an **Internal Entity** starts and it try to establish a connection with other entities (for example if this entity is a **Naming Servant**, it will try to establish a connection with the **Naming Request Broker** and the other entities that it needs)
- if it's impossible to establish a connection with other entities, this entity will restart from the step relating to the trying to establish the connection and it will communicate the **establish connection failure** to the **Outside Interfacing Layer** (composed by the **Request Handler** entity described in the chapter relating to the module's structure). That entity will manage all issues relating to the internal connection unavailability (for example it can save the pending requests in a cache in order to forward them to the internal entities when all the internal connections will be established)
- if the connection has been successfully established, the entity will run normally as described in the **Internal Entities life cycle**

The described working steps are valid for any **Internal Entity**, so it is valid both if it's a **"father" entity** as it's a **"child" entity**.

The decision to adopt this paradigm derives from the need for reliability because with this interaction model, if there is a failure relating to the unavailability of the connection between the internal entities, the **Outside Interfacing Layer** can manage the critical situation.

Then, if there is a connection problem between the internal entities, the **Request Handler** can maintain all pending requests in a cache in order to forward them when the module will be able to work normally (like described in the previous list).

5.7 Fault Tolerance Case Study: Unavailable Network Connection

This section contains all details about the **fault tolerance case study** relating to the **network connection unavailability**.

This is a very important issue because if there is no **network connection** (this term is related to the *local connection* and the *internet connection*), this module can't work. This is an important aspect because, like described in the previous chapter, this module is vertical layered and distributed (each layer works separately) but it can be also distributed in many parts based on the **network distribution**.

This model has been designed basing on the main concept adopted in the interaction model about the connection unavailability between the internal entities.

The introduced model will be shown in the following picture.

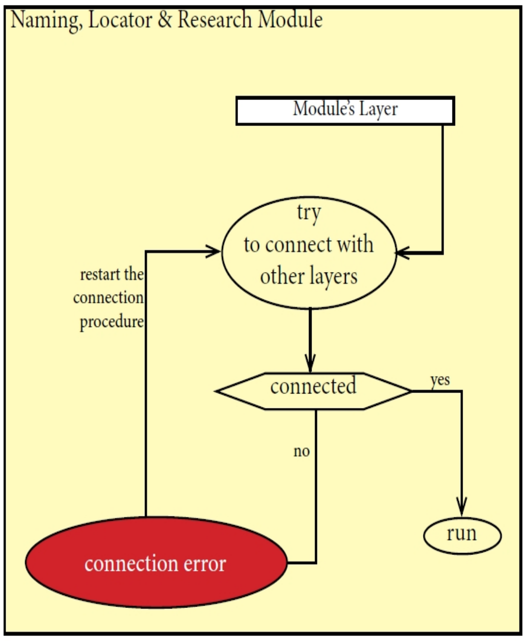

Naming, Locator & Research Module
Module's Layer
try to connect with other layers
restart the connection procedure
connected
yes
no
run
connection error

Figure 5.22: Fault Tolerance Unavailable Network Connection Case Study Interaction Model

All problems and issues due to the unavailability of the network connection can be solved by restore of the network connection but this is not an aim of this module. The only operation that this module can do is retrying to establish the connection.

As you will see in the following sections, the connection to the network is independent from this module. In each network connectivity problem if the problem can't be solved, this question will be notified to the **consumer entity** in order to inform it that there is a critical situation and that the network connectivity can't be restored.

All interactions that will be described are valid for any part of this module but the fault tolerance complete working steps depend on two different case: the **temporary unavailable network connection** and the **persistent unavailable network connection** that will be described in the following sections in order to give a specific fault tolerance solution for any critical situation related to this issue.

5.7.1 Fault Tolerance Case Study: Temporary Unavailable Network Connection

This is the case study relating to a **temporary unavailability of the network connection**.

In this case the detection of the connection unavailability is aim of the **Outside Interfacing Layer** because it has to communicate to the **consumer entity** that the network is unavailable.

In this situation there are a definend numer of tries to restore the network connection. If a try to restore has success, the restored network connection state is to be communicated to the **consumer entity** in order to notify to it that the module can run normally.

The main interaction model will be shown in the following picture.

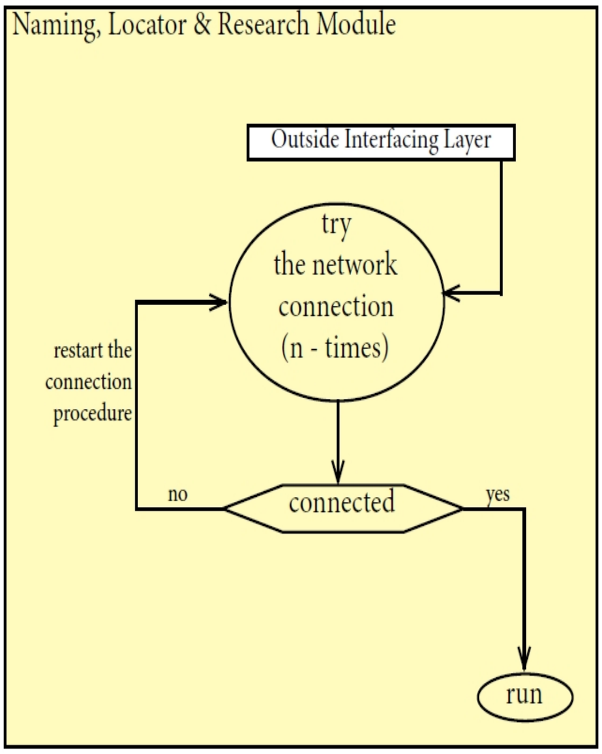

Naming, Locator & Research Module
Outside Interfacing Layer
try the network connection (n - times)
restart the connection procedure
no
connected
yes
run

Figure 5.23: Fault Tolerance Temporary Unavailable Network Connection Case Study Interaction Model

In details, the working steps are the following.

- the **Outside Interfacing Layer** (composed by the **Request Handler** entity) try the network connection. This procedure will be done for a defined number of times (signed with n in the above picture)
- if the current try has **success**, the module will run normally without problems and the module working will start (or continue if the critical situation is after the module starts)
- else, if the try has **no success**, the **Request Handler** entity will restart the connection procedure described in the previous list item

If the number of attempts reaches the defined massimum number (the number **n** showed in the previous picture), the **network connection** can't be restored, so you will have the situation related to the next section: the **Persistent Unavailable Network Connection Case**.

However, if the network connection is unavailable, all entities and all parts of this module won't run until the network problem will be solved because each layer of this module needs a connection to the other layers.

5.7.2 Fault Tolerance Case Study: Persistent Unavailable Network Connection

This is the case study relating to a **persistent unavailability of the network connection**.

This situation derives from the previosly seen case study because if the defined number of trying (described in the previous section) has no success, the network becomes marked as **persistent unavailable** so, this module can't run.

Logically, this situation is related to the solution of the connectivity problem that was harnessing the network connection.

After the solution to the described problem, the module has to be restarted in order to check if any problem is solved. If everything works correctly, the module will run normally, else, the module won't work so this procedure has to be repeated until the module detects the **correct network connectivity** in order to work normally.

The main interaction model will be shown in the following picture.

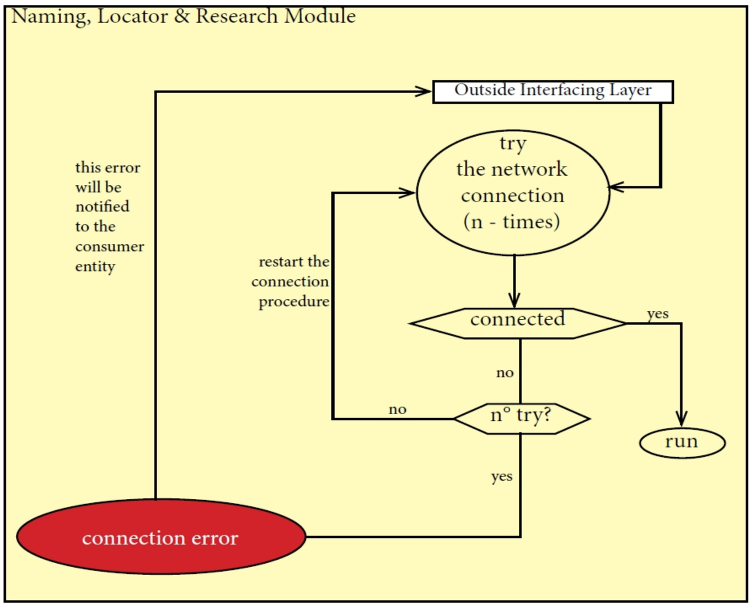

Naming, Locator & Research Module
Outside Interfacing Layer
try the network connection (n - times)
this error will be notified to the consumer entity
restart the connection procedure
connected
yes
no
no
n° try?
run
yes
connection error

Figure 5.24: Fault Tolerance Persistent Unavailable Network Connection Case Study Interaction Model

In details, the working steps are the following.

- the **Outside Interfacing Layer** (composed by the **Request Handler** entity) try the network connection. This procedure will be done for a defined number of times (signed with n in the above picture)
- if the current try has **success**, the module will run normally without problems and the module working will start (or continue if the critical situation is after the module starts)
- else, if the try has **no success**, the number of attempts already done will be checked. If the attempts' number is lesser than the maximum (signed with **n** in the above picture), the **Request Handler** entity will restart the connection procedure described in the previous list item
- else, so the total number of the attempts is greater than the defined maximum value, the procedure to try the network connection will be stopped and a **connection error** will be notified to the **consumer entity**. This notification step is an aim of the **Outside Interfacing Layer** because it is the layer that can communicate with the **consumer entity** without network connection (these entities are on the same device)

However, if the network connection is unavailable, all entities and all parts of this module won't run until the network problem will be solved because each layer of this module needs to the connection with other layers that, like described in the previuos section, are network connected.

5.8 Fault Tolerance Case Study: Server Updating Error

This section contains all details about the **fault tolerance case study** relating to the **error events during the server updating phase**.

Each **Server Updating Error** is due to data updating process (the server system is based on Cassandra , as seen in the first chapter) and it generates an error due to a **database updating error**, so in this case the updating of a data on the database generates an error.

This is a foundamental case study because each function of this module can't work correctly if the **Storage Layer** entities don't work normally, so if there are many problems relating to the **servers updating operations**.

This issue is composed by many factors, basing on the nature of the problem but the only kind of problem related to this specific case study is that due to the **database updating error** because all other aspects (for example if a server entity crashes, etc) have been treated in the previous sections about the **fault tolerance**.

The main interaction model of this case is shown in the following picture.

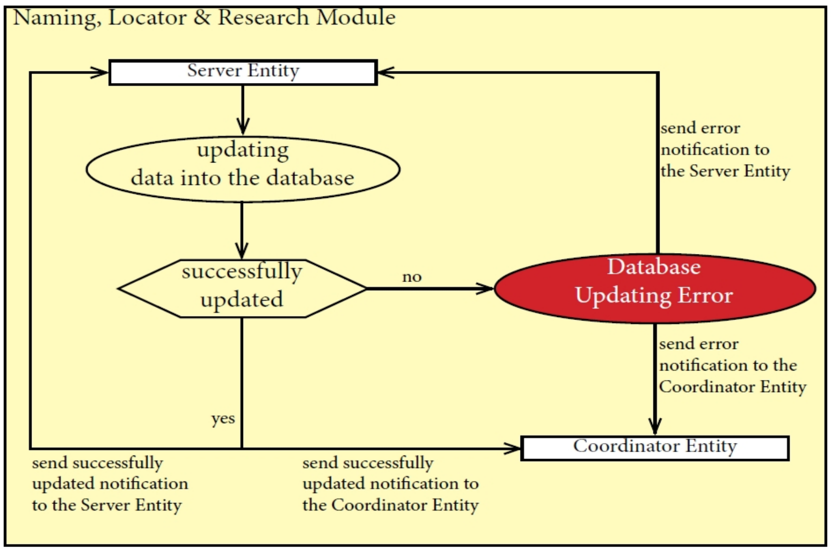

Naming, Locator & Research Module
Server Entity
updating data into the database
successfully updated
no
Database Updating Error
send error notification to the Server Entity
send error notification to the Coordinator Entity
yes
Coordinator Entity
send successfully updated notification to the Server Entity
send successfully updated notification to the Coordinator Entity

Figure 5.25: Fault Tolerance Updating Database Error Case Study Interaction Model

In details, the working steps are the following.

- the **Server Entity** that has received a **data update request** starts to update the database
- if all target data have been successfully updated, the **Database System** sends a notification to the **Server Entity** and **Coordinator Entity**; Logically, the **Coordinator Entity** will notify to other **Server Entities** to update the target data (this process is not contained in the above picture because it has been described in the previous chapters)
- else, so if any part of the target data has not been successfully updated, a **Database Updating Error** is detected and the **Database System** sends a notification to the **Server Entity** and **Coordinator Entity**; This notification process is needed in order to give to other entities of the **Data Storage Layer** all informations about the store process state. When the **Server Entity** receives this error notification it will restart the updating process in order to retry to update the received target data

In the above description we have introduced the concept of **Database System** because this is the internal computation part of the data store system.

This system is the set of operations and entities that have the aim to manage the phisical store of the target data (for example an entity of this system is the database entity able to update the target column related to the received data).

All details about the **Database System** are related to the Cassandra 's model because the complete model about **Data Storage Layer** is based on that model (it has been described in the first chapter related to the background knowledge).

Chapter 6

Future Developments

All details about the future developments of the treated module will be explained in this chapter.

The first future developments that will be described is the development relating to the possibility of having an **Individual Tuple Space** so, a **Tuple Space** for each active entity (the *agents*) that use this module; this development is very important because this characteristic allows to all entities to communicate to each **entity's Individual Tuple Space**.

The second and last future development that will be introduced is the **customized query language** about the **Data Storage Layer**. This development is very important because it will increase the efficiency of the data store operations; this more efficiency is given from the lower response time of each database operation. This development is possible because Cassandra allows to use a **custom query language** for all operations, so basing on the needs it's possible to create a new query language in order to increase the data store system efficiency (like *Facebook* where the adopted query language is a custom one, the *FQL* that stands for *Facebook Query Language*).

All these future developments will be treated and described in details in the following sections.

6.1 Individual Tuple Space

This future development is very important because this module allows to each active entity (*agents*) to communicate directly with another one without a **central Tuple Space**, so all *agents* will have an **Individual Tuple Space**.

Given these needs, the abstract syntax of an operation *op* can be invoked on the destination active entity who has the name *eid* is:

[**enid ? op**]

So, using that syntax, the operation *op* will be made on the **Individual Tuple Space** of the target entity who has the name *eid*.

This syntax doesn't replace the previous one introduced in the first chapter because this one integrated with the previous one and joined with the new naming syntax will increase the **system functionalities**.

An example of use of this new type of **Tuple Space** is shown in the following picture where an entity make an operation on the **Individual Tuple Space** of another entity, the target entity.

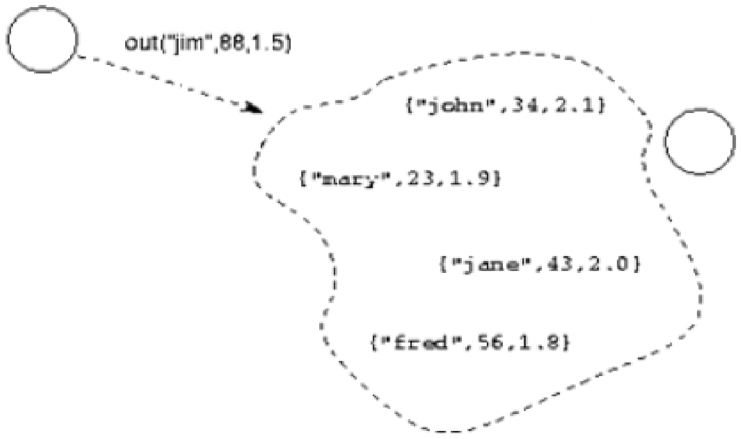


Figure 6.1: Example of use of an individual tuple space.

In the end we have to explain that this future development will increase the system efficiency and it will improve all interactions and operations between all the **consumer entities** because it will make possible to have a sort of direct message passing between all entities.

6.2 Customized Query Language

This is the second and last future development that we will introduce.

This is a very important development because nowadays one of the most important problem in *distributed system* is related to the data store operations efficiency, so this development is related to the **Data Storage Layer** because this layer contains all entities relating to the *data store* and the *database*.

This development is possible because Cassandra (the **Distributed Data Storage Model** adopted in this module) allows to adopt a *custom query language* in order to allow to the developers to create a query language ad-hoc to satisfy all requirements of the designed system.

Then, we have decided to explain this future development because with this **custom query language** it's possible to increase the efficiency and decrease the response time of all operations that the entities make on the database. This improvement is given by a foundamental important factor: if there is an ad-hoc query language, all requests can be less complex because the data store system needs only to receive the data and the operation type (for example update, replace, etc.) without any other information that in many cases is an overhead data.

One of most important custom query languages introduced in the last years is the *FQL* that stands for *Facebook Query Language*; this query language has been introduced during the *Facebook* development because this service adopts a *not relational database model* like the database model adopted in this module.

The *Facebook* developers were having the problem of the too high response time to database operations so, they decided to adopt a new data store model based on Cassandra . That model allows to adopt a custom query language, so the developers decided to create a new one in order to increase efficiency in accessing to the database and decrease the response time to all operations.

Then, after the study of this query language and the main problems that this module's **Data Storage Layer** has in common with the *Facebook Data Store Model* we have decided to advise this one as an important future development.

Chapter 7

Conclusions

All conclusions about this thesis will be described in this chapter in order to give a complete view of the work done.

First of all, we have studied all details about the model, architecture and technology of TuCSoN because it is the considered coordinated consumer middleware for **LoReNa**. This does not mean that the designed system is TuCSoN -specific because, as described in the previous chapters, this system can be adopted for any kind of middleware as consumer. The reason of the choice about TuCSoN as consumer derives from the need to give a complete naming system to that middleware updated with the nowadays technologies.

After the study about TuCSoN , we have decided to analyze **JADE** because it has a model and structure that can be adopted in this module. Starting from **JADE** we have designed a new model that can be more efficient and oriented to the network distribution. Then, we have designed the model of **LoReNa** that has many feature in common with **JADE** but that it is more extendible and dynamic than the other one because the studied one has many features that allow to distribuite the module on the network in many different ways; for example **LoReNa** can be distributed both dividing one or more layer from the others both as a vertical layer division as in an orizontal layer division without problems about interactions or reliability. This is a very important characteristic of it because this different division possibilities make **LoReNa** compatible with any kind of device regardless of the power and computational capacity.

After these cited studies, we have designed the structure of the two upper layers: the **Outside Interfacing Layer** and the **Internal Computational Layer**. That design has been made in order to obtain all distribution and dynamic features. During this phase we paid attention particulary to all different distribution cases and interactions, so the design has been made in a very detailed way in order to give a complete useful description from many viewpoints.

The last layer designed is the lower one: the **Data Storage Layer**. This layer is not used by all services but only by two services: the Naming Service and the Locator Service. The reason is that the Research Service does not need of a data store service because the functionalities of that service are related to the communication with the other services. During the design phase of this layer we have decided to study a particular data store model: Cassandra . This decision has been made because that model is very efficient and used by many internet services (the most famous one is *Facebook*), so it is a great choice because an important requirement given by the initial study of **LoReNa** was the need to have a *distributed data store system*.

After the design of all layers we have studied and defined a new naming syntax: the **Universal Naming Syntax**. We have studied this syntax in order to provide a universal one, based only on the utilization of an **Universal Identifier** that can provide an unique name to any consumer entity. We have also defined all procedure steps that an entity has to do to obtain an identifier.

In the final part of the case studies chapter we have explained all case studies relative to **LoReNa** dividing them by type; in fact, the main division that we made is between the service type: Naming Service, Locator Service and Research Service. After this division we made a further internal case study division by the distribution case: Local and Distributed. These divisions have been foundamentals in order to explain all details about the interactions and communications between all entities involved in all services working.

The Fault Tolerance case studies have been explained in details in the last part of the case studies in order to give a detailed description of all policies and procedures to solve all issues related to faults, errors or any kind of problems.

Logically we have explained only the cases relating to the main fault cases (for example related to the crash of any entities, etc.) because it's impossible to give a solution to any kind of problems during the design phase because many problems are related to the technology or to the environment, then they will be treated during the implementation and testing phase.

The last chapter of this thesis contains the description of the most important future developments of **LoReNa** in order to give a guideline to many aspects that can improve its functionalities and characteristic in terms of efficiency, optimization and performance. In fact, we have described the two most important future development: the **Individual Tuple Space** and the **Customized Query Language**.

At last we have to give the description of another important aspect of **LoReNa**: the high **compatibility** of this module with any consumer middleware. Thanks to the dynamic and extendible structure of this module, it is able to adapt itself to any middleware because the only entities involved in this changing are the **Servant Entities** because the lower entities are completely independent from the consumer middleware and communicate with it through these **Servant Entities** who has the aim to check and forward any received request.

In the end, after all the descriptions of **LoReNa** we can define it as a complete, dynamic and extendible module that can be improved by the described future developments. It can be also implemented in many programming languages and adopted in many various system with any middleware as consumer.

Ringraziamenti

Grazie ai miei genitori e a tutti i miei parenti che mi sono stati vicini in questi tre anni di impegno assiduo e continuo. Per il loro supporto e la loro spinta a continuare e non arrendermi mai, davanti a qualunque imprevisto e problema. Ringrazio particolarmente i miei genitori perchÃ¨ mi hanno dato la possibilita' di focalizzarmi a tempo pieno nello studio grazie ai loro sacrifici. Spero di averli ripagati almeno in parte per la fiducia e il supporto datomi in tutti questi anni.

Grazie a Giulia, la mia fidanzata, che ha avuto la forza di starmi accanto in ogni momento in questi anni, sopportando anche i momenti in cui lo stress che mi affliggeva di continuo e le preoccupazioni che a volte mi assalivano. Grazie soprattutto della spinta a dare sempre il mio massimo, senza arrendermi mai, e a sforzarmi sempre piu' per riuscire il meglio possibile negli studi. Grazie davvero di cuore.

Grazie a Francesco, un grande amico con cui ho passato momenti devastanti durante vari progetti per alcuni esami. Grazie soprattutto per i preziosissimi consigli professionali e non, datemi durante questi anni e per i momenti di relax e divertimento che non sono mancati per fortuna.

Grazie a tutti i miei amici che mi hanno aiutato in questi anni a divertirmi e svagarmi nei momenti di non-studio, per le serate passate insieme a divertirci. Per tutti quei momenti di gioia che mi hanno regalato anche nei periodi di ansia tremenda che mi assaliva per gli esami che dovevo dare.

Grazie ai miei compagni di corso e a tutti coloro che ho conosciuto e con cui ho allacciato un rapporto di amicizia durante questi tre anni. Grazie per i momenti di relax, per i momenti divertenti e per i momenti di ansia e stress, per i momenti di sostegno che davamo ognuno all'altro durante lo studio e i progetti.

Grazie al mio relatore, il Prof. Andrea Omicini, e al mio correlatore, l'Ing. Stefano Mariani, che mi hanno supportato durante questo interminabile periodo di tesi. Grazie per avermi anche sopportato durante le innumerevoli pagine scritte e per il supporto datomi durante i vari ricevimenti per concordare le vie da seguire per lo sviluppo della tesi.

Grazie ai professori che mi hanno insegnato molto durante questi tre anni, che mi hanno spinto a non arrendermi mai e ad impegnarmi sempre di piu' per passare tutti gli esami che ho sostenuto, che non sono di certo una passeggiata ad ingegneria.

Bibliography

[1] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun.*, ACM, 35(2):97-107, 1992.

[2] Andrea Omicini. On the semantics of tuple-based coordination models. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187-200. CRC Press, Boca Raton, FL, USA, October 2002.

[3] Andrea Omicini. Towards a notion of agent coordination context. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 175-182, San Antonio, TX, USA, 28 February - 2March 1999. ACM. Special Track on Coordination Models, Languages and Applications.

[4] Peter Wegner. Coordination as constrained interaction. 1996.

[5] Peter Wegner. Why interaction is more powerful than computing. *Communications of the ACM*, 40(5):80-91, 1997.

[6] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87-119, 1994.

[7] Paolo Ciancarini and Chris Hankin. Coordination Languages and Models, First International Conference *COORDINATION '96, Cesena, Italy, April 15-17, 1996, Proceedings, volume 1061 of Lecture Notes in Computer Science*, Springer, 1996.

[8] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Coordination technologies for internet agents. *Nordic J. of Computing*, 6(3):215-240, 1999.

[9] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, David Rossi, and Andreas Knoche. Coordinating multiagent applications on the www: A reference architecture. *IEEE Trans. Softw. Eng.*, 24(5):362-375, 1998.

[10] David Gelernter and Paolo Cianciarini. A distributed programming environment based on multiple tuple spaces. *International Conference on Fifth Generation Computer Systems*, pages 926-933, 1992.

[11] David Gelernter and Paolo Cianciarini. A distributed programming environment based on multiple tuple spaces. *In Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 926-933, Tokyo, Giappone, 1992. Institute for New Generation Computer Technology.

[12] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277-294, November 2001.

[13] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169-177, Atlanta, GA, USA, 27 February-269 1 March 1998. ACM. Special Track on Coordination Models, Languages and Applications.

[14] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251-269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[15] Andrea Omicini and Enrico Denti. Formal ReSpecT. *Electronic Notes in Theoretical Computer Science*, 48:179-196, June 2001. Declarative Programming - Selected Papers from AGP 2000, La Habana, Cuba, 4-6 December2000.

[16] Andrea Omicini and Franco Zambonelli. Coordination of mobile information agents in TuCSoN. *Internet Research*, 8(5):400-413, December 1998.

[17] John W. Lloyd. *Foundations of Logic Programming, 1st Edition*, Springer, 1984.

[18] Antony Ian Taylor Rowstron. *Bulk Primitives in Linda Run-Time Systems*, PhD thesis, The University of York, 1996.

[19] C2 webpage. Tuple Space. `http://c2.com/cgi/wiki_TupleSpace`.

[20] Wikipedia webpage. Tuple Space. `http://en.wikipedia.org/wiki/Tuple_space`.

[21] Argonne national laboratory webpage. Tuple Space. `http://www.mcs.anl.gov/itf/dbpp/text/node44.html`.

[22] APICe Webpage. ReSpecT. `http://apice.unibo.it/xwiki/bin/view/ReSpecT`.

[23] Andrea Omicini and Stefano Mariani. The TuCSoN coordination model and technology: a guide. `http://www.slideshare.net/andreaomicini/the-tucson-coordination-modeltechnology-a-guide`.

[24] UUID: Universally unique identifier javadoc. `http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html`.

[25] MongoDB Web Site. `http://www.mongodb.org`.

[26] Dynamo Web Site. `http://aws.amazon.com/dynamodb/`.

[27] Cassandra Web Site. `http://cassandra.apache.org/`.

[28] RDBMS Web Site. `http://en.wikipedia.org/wiki/Relational_database_management_system`.

[29] Google's BigTable Wikipedia Webpage. `http://en.wikipedia.org/wiki/BigTable`.

[30] Jade Web Site. `http://jade.tilab.com/`.

[31] FIPA Web Site. `http://www.fipa.org/`.

[32] Multi-Agents System Wikipedia Webpage. `http://en.wikipedia.org/wiki/Multi-agent_system`.

[33] Peer to Peer Wikipedia Webpage. `http://en.wikipedia.org/wiki/Peer-to-peer`.

[34] Cloud Computing Wikipedia Webpage. `en.wikipedia.org/wiki/Cloud_computing`.