Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

MSC Dissertation in Computer Engineering

1st Semester, 2008/2009

Orchestration of Heterogeneous Middleware Services and its Application to a Command and Control Platform

Paulo Cancela

Nº26368

Supervisor

Prof. Doutor Hervé Miguel Cordeiro Paulino

February the 20th, 2009

Nº do aluno: 26368

Nome: Paulo Filipe Neves Bento Cancela

Título da dissertação: Orchestration of Heterogeneous Middleware Services and its Application to a Command and Control Platform

Palavras-Chave:

- Integração de sistemas
- Arquitecturas Orientadas a Serviços
- Computação Distribuída
- Arquitecturas de software de Comando e Controlo
- Orquestração de serviços

Keywords:

- System integration
- Service-Oriented Architectures
- Distributed Computing
- Command and Control software architectures
- Service orchestration

Resumo

Até há pouco tempo, os objectos distribuídos têm sido a tecnologia líder no desenho e implementação de arquitecturas baseadas em componentes ou serviços, normalmente conhecidas por Arquitecturas Orientadas a Serviços. No presente, devido ao seu alto desempenho, aspecto fundamental no contexto das plataformas, esta tecnologia continua a desempenhar uma função importante no desenvolvimento de sistemas distribuídos. No entanto, apesar de já estar estabelecida no mercado há mais de uma década, apresentando-se, portanto, mais consistente, falhou na transposição do conceito de Arquitectura Orientada a Serviços para a *Web*.

Os *Web services* são uma tecnologia recente que tem amadurecido nos últimos anos. A sua aceitação no seio das empresas e organizações tem vindo a crescer, uma vez que ultrapassa os problemas dos objectos distribuídos, tais como a interoperabilidade e transposição para a *Web*. A interoperabilidade entre sistemas é o ponto forte desta tecnologia, uma vez que este é um aspecto crucial para a logística de negócio nos dias de hoje. Além disso, a utilização de serviços trouxe um novo conceito: composição de serviços, uma técnica de criação de novos serviços. Contudo, apresenta-se ainda muito vocacionada e dependente de *Web services,* não havendo ferramentas que suportem o conceito para outras tecnologias.

Para que seja então possível usar a composição de serviços em plataformas baseadas em objectos distribuídos, é necessário expôr os seus serviços como *Web services,* aparecendo assim o principal objectivo desta tese de mestrado: fornecer suporte para a composição de serviços originários de plataformas baseadas em objectos distribuídos. Uma vez que estas plataformas são normalmente compostas por bastantes serviços, a ideia deste estudo é apresentar uma plataforma como um conjunto de *Web services,* de forma a possibilitar a aplicação do conceito de composição de serviços, através da técnica específica de orquestração.

Abstract

Distributed objects was, until recently, the leading technology in the design and implementation of component-based architectures, such as the ones based on services, better known as Service-Oriented Architectures (SOA). Although established in the market for more than a decade, and therefore mature, these technologies have failed to overcome the porting of the SOA concept to the Web.

Web services are a recent technology that has been growing in the last few years. Their acceptance has increased over enterprises and organizations as they seem to overcome the Web and interoperability related problems of the Distributed Objects technology. Web services provide interoperability between systems and that is undoubtedly a strength of this technology since this is a crucial aspect of nowadays business. Moreover, the widespread of services led to the recent introduction of the service composition concept, that although being a technology independent concept, is closely related to Web services and there is no tool support for other technologies.

Nonetheless, distributed objects still play an important role in the development of distributed systems, namely due to performance issues that are important when it comes to the internals of a platform. However, the use of service composition in these distributed object-based platforms requires the exposure of their composing services as Web services.

The main objective of this masters thesis is improve the state-of-the-art in the support for the composition of services originating from distributed objects-based platforms. Bearing in mind that these kind of platforms are composed by several services, the idea is to present a platform as a set of Web services in order to be able to orchestrate them.

Acknowledgements

This work was developed with the support of Critical Software S.A, that is contributing with an internship locally supervised by Eng. Tiago João Parreira da Gama Franco.

I would like to acknowledge all my colleagues, especially Tiago Franco, from Critical Software for the great support given in my integration within Critical Software. Their great suggestions and technical support were a huge contribute for the development of this work. Thanks also to Critical Software S.A. for having supported my internship almost over a year.

I must also express all my sincere gratitude for my supervisor Hervé Paulino, who were always available to support me with his skills and knowledge. Moreover, his assistance during both preparation and development phases, including the technical reports were vital in the conclusion of this masters thesis.

A very special thanks goes to my family, particularly to my mother, Ana Maria Cancela, my brother Miguel Cancela and my father José Cancela, for the encouragement to perform this masters thesis and for all support they provided me through my all life.

Last, but not least, my deep thanks to my girlfriend Ana Raquel Martins, since it would be very difficult for me to finish this thesis without her motivation, friendship and love. Special thanks for putting up with me all these years.

Index

Figure Index

Table Index

1. Introduction

1.1. Motivation

Distributed objects was, until recently, the leading technology in the design and implementation of component-based architectures, such as the ones based on services, better known as Service-Oriented Architectures (SOA). Examples of such technologies are the Common Object Request Broker Architecture (CORBA) **[2]**, a standard defined by the Object Management Group (OMG) and the Distributed Component Object Model (DCOM) **[3]** from Microsoft. Although established in the market for more than a decade, and therefore mature, the distributed object technologies have failed to overcome at least two aspects that are very important to the nowadays' businesses: the porting of the SOA concept to the Web, mainly due to the inability to pass through firewalls, and the support for interoperability between different technologies. Also, distributed object architectures follow a tightly coupled approach, fact that can be seen as a drawback, since it requires, in most cases, the definition of dependencies between the components of the system.

To tackle these issues, the Web services technology is being increasingly adopted by organizations and enterprises. This is mainly due to the fact that having the SOA concept through the Web, promotes business visibility, and the interoperability provided between different technologies, is a crucial aspect to nowadays business, particularly in business-to-business transactions. The Web services technology has been around since the late nineties, but it was only in 2002 that W3C published its architecture **[25].** The appearance of standards and of the Web Services

Interoperability Organization (WS-I)[1], that focuses and drives only on Web services, has contributed to the popularity increase of this technology.

Nevertheless, distributed objects are still a major player in the SOA realm. They are an established technology used in the development of many platforms, which results in a more mature and better *performant* support than Web services.

Currently, many SOA-based software is built on top of distributed objects, which disables the possibility of Web integration. The porting of these software architectures to the Web services technology is usually not viable due to the cost and effort it represents. Moreover, most of the times, the overhead introduced by platform and language independence provided by Web services is not desired when it comes to the internals of a platform.

One other fundamental aspect is that, although both distributed objects and Web services are distributed systems technologies, have a kind of definition interface, and provide similar resources for component registration and discovery, they use different programing paradigms. While distributed object technologies are object-oriented, Web services technology is document-oriented, since it is based on the exchanging of XML documents and does not have the concept of object. The truth is that the Web service technology was designed to support Web access and not to replace distributed object technologies.

We have thus to conclude that porting existent distributed object-based platforms to the Web is, in fact, a costly and not trivial operation. But, on the other hand, moving to the Web services world opens a new range of prospects, essentially motivated by: increased visibility, the business becomes accessible from the Web; business-to-business interaction based on XML standards; and the use of service composition to deploy new services by composing platform and other Web available services. In fact, the last two are closely related, since service composition plays a major role on the support of business-to-business and of enterprise application integration. The use of

[1] http://www.ws-i.org

third-party services on the definition of a platform's own business model is becoming a common solution.

However, service composition is essentially driven by interoperability, and therefore, it is only natural that existent solutions **[18, 22]** are restricted to Web services. Thus, in order to use this concept in the context of distributed object platforms, it is necessary to expose their composing services as Web services. This exposing process is also known as bridging.

1.2. Bridging Distributed Objects

Bridging distributed objects is a process that consists on exposing a service provided by one of those technologies as a Web service. This makes the service available for the Web and enables its interaction with different technologies, since interoperability is one of the main arguments of the Web services. Moreover, a bridged service can be composed using the available Web services composition tools.

Figure 1: Bridging process illustration

For a better understanding of this subject, Figure 1 portrays what we should expect from a bridging process. The main idea in this context is to allow a Web service client to invoke a service based on a distributed object technology in a completely transparent way, i. e., as if it was invoking a Web service. As it is demonstrated in the figure, a Web Service will act as a proxy between the server and the client. Both client and server interact each other with their own standards, leaving the conversion of the

messages for the proxy, fact that provides to the process a transparent nature.

1.3. Service Composition in Distributed Object Technologies

As stated previously, technologies like CORBA, DCOM or RMI used to be the distributed objects market leaders, but lost that position with increased importance of Web access, particularly in business-to-business interaction. So, if Web services are currently more attractive than these technologies, why just not re-implement the existing platforms? The answer lies in the associated cost, namely when it comes to widely used and accepted platforms. A Web service based re-implementation might represent an huge investment of both time and money. Moreover, the lack of Web services support for some of the usual distributed-objects' features, such as event-handling, may even force some architecture re-design instead of just code re-writing.

One other factor in favor of distributed object technologies is performance. Web services do not behave as good as distributed objects performance wise. This is mainly because of the overhead associated to the platform and language independence, one of the great advantages of this technology. Even when it comes to the design of a new distributed system, the trade off between the interoperability provided by Web services and the higher performance provided by distributed object technologies should be considered.

An attractive solution would be to integrate an existing distributed object platform with Web services, which is possible by exposing the services of the former as Web services. This solution protects the investments made in the past, while allowing the stepping in the Web world.

To the best of our knowledge, there are no solutions to address the bridging and consequent composition of distributed objects platforms, and the current state-of-the-art of bridging distributed objects services as Web services is very limited. All approaches **[4, 14, 17, 32, 33]** focus on the bridging of a single service, are designed to a single technology and require the posterior publishing of the Web service, thus not

providing a systematic and transparent procedure.

From the current state-of-the-art, Apache Axis2[2], a Web services engine that provides a module that allows bridging CORBA services, is the solution that better suits our purpose. However, the bridging process is not platform-oriented, is bound to CORBA and it implies a manual procedure to bridge the services. A XML file has to be created, with some information regarding the service and together with the corresponding IDL interface, has to be stored in the directory where the module is installed. This process is repeated for each service to bridge.

This state-of-the-art lead us to develop **OHMS (Orchestration of Heterogeneous Middleware Services)**, a platform for the systematic exposure of a distributed object-based platform as a set of Web services. The main idea is to expose services from a distributed object-based platform in a systematic way, in order to allow the definition of new services composing them through orchestration.

OHMS supports, thus, the orchestration of services from distinct distributed objects technologies. It is platform-oriented, meaning that it does not expose a single service, but rather a platform. These expose their set of services partially or completely by registering their name-servers in OHMS. No alterations on the original platform's code are required. Transparency is one of our main premises.

The OHMS platform is composed of two components (figure 2): the **Name Service Directory module** (or simply **directory**) and the **Orchestration module**. The directory provides an interface in order to accept registrations of logistics for bridging distributed objects technologies and of the name-servers. The orchestration module consists in the extension of an existing orchestration tool, the Eclipse BPEL plug-in[3]. Bridged services are registered in a UDDI repository embedded in the directory. This repository binds both components, since the orchestration module will be able to access and orchestrate the services it stores. The architecture of OHMS will be fully

[2] http://ws.apache.org/axis2

[3] http://ww.eclipse.org/bpel/

described in chapter 3.

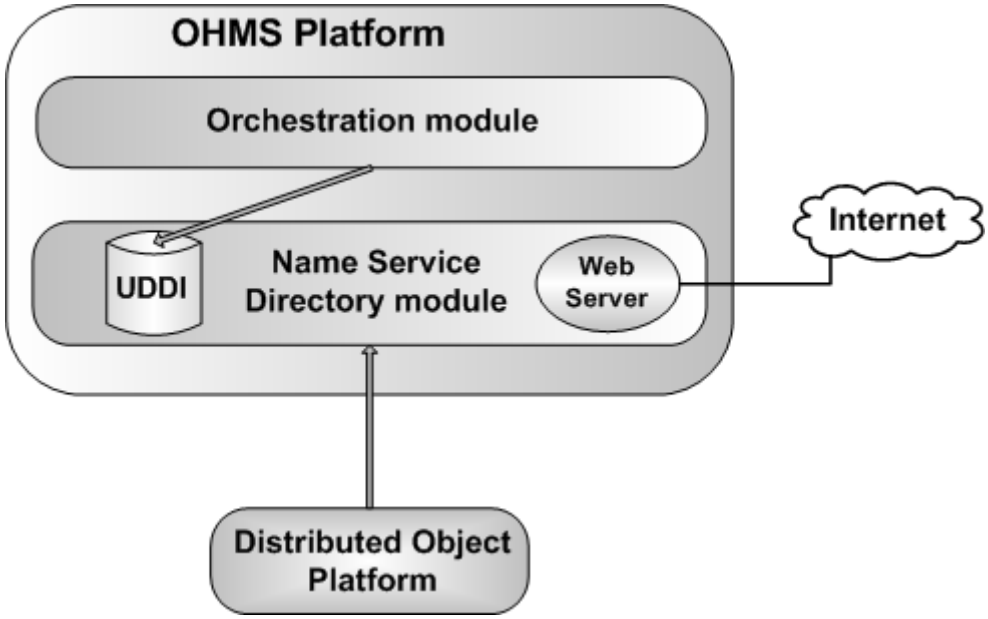

Figure 2: The components of the solution

Services from the original platforms register in their name-servers as before. OHMS will be responsible for both their bridging and publishing (in the UDDI repository) making them visible to the orchestration module. To this module, bridged services are orchestrated as common Web services. The only difference lies in the that they these are selected by browsing a previously imported platform. This is a very useful and intuitive functionality to orchestrate services originating from different directories, in order to achieve platform interoperability.

1.4. Application to a Command and Control Platform

Another motivation for this work arose in the context of the Command and Control Platform (COMCOP) developed by Critical Software. COMCOP was developed on top of the CORBA technology and has the ambition of creating new services through orchestration, using the existing ones and also Web services available in the Web, thus providing a perfect test-bed for OHMS.

Services in COMCOP's architecture are distributed in layers. The core layers include general, solution independent, services and solution specific layers include the services that specify the platform for a given assigned mission. Composing a new service by resorting to the existing ones is an attractive solution for this platform. A concrete example is a Mission Planning Service with the ability of defining a specific plan for each mission. This is an example of a service that would be composed entirely of other platform services. However, other examples may use services provided by an Web entity external to COMCOP, like the weather forecast for example.

For Command and Control solutions of the same kind of COMCOP, the concept of orchestration is certainly very attractive. However, the state-of-the-art does not provide suitable support to the application of this concept to platforms developed on top of distributed object technologies. In this context, the solution to use OHMS to expose these distributed objects platforms as sets of Web services in order to orchestrate their services is, in our opinion, the best approach.

1.5. Contributions

The main objective of this masters thesis is to provide support for the composition of services, originating from distributed objects-based platforms. Bearing in mind that these kind of platforms are composed by several services, the idea is to present a platform as a set of Web services. The contribution we envision are:

1. The design and implementation of a platform that provides the systematic exposing of services from distributed objects-based platforms as Web services.
2. The extension of the Eclipse BPEL plug-in to provide the possibility to connect to distributed object-based platforms and include those exposed services, referred in the previous point, in a BPEL Process. This include the possibility of connection to one or more repositories.
3. Application of the OHMS platform to the world of command and control platforms, concretely to COMCOP. This requires the specification of OHMS to handle the bridging of CORBA platforms, which can also be seen as a

contribution.

1.6. Outline of the thesis

This remainder of this thesis is structured as follows. Chapter 2 presents a description of the state-of-the-art on the areas of distributed objects, describing the main features of some existing distributed objects technologies, and of Service-Oriented Computing, describing the concepts of component architectures and Web services. We also focus on the Service Composition concept and its available techniques: orchestration and choreography. Finally we introduce bridging technologies currently available for bridging distributed objects technologies and Web services.

Chapter 3 describes the core of the work developed in this thesis. We will start by focusing on the OHMS architecture and then we will present an extensive description of both directory module and orchestration module.

Chapter 4 introduces the concept of Command and Control platforms, presents COMCOP, our case study, and the procedure required to use OHMS in the orchestration of COMCOP services, namely the support for bridging CORBA-based systems.

Finally, Chapter 5 concludes the work developed and includes some suggestions for the future developments.

2. State-of-the-Art

This chapter presents an overview of the subjects of distributed objects, service-oriented computing, service composition, and the pf bridging distributed technologies. A good knowledge of them is necessary to fully understand the solution proposed in this thesis.

The chapter is divided into four major sections. The first section presents an overview of the distributed object concept including some concrete examples, used in the software solutions crucial for this work. The second one introduces the concepts of Service Oriented Computing and Web services. These are the key concepts for service composition through Orchestration and Choreography, which is described in the third section. Last section, explains the concept of bridging Web services and distributed objects referred in the previous sections.

2.1. Distributed Objects[4]

A system is considered distributed when there are several components available over various machines and these components need to interact with each other. We have the notion of distributed objects when each one of these components is seen as an object**.**

Distributed objects are software modules that can be distributed among different machines with different platforms and can communicate in a transparent way, i. e., as if they were running in the same machine. Each of these modules is considered an

[4] This section is mostly based on the book Engineering Distributed Objects **[5]**.

object, and the communication is processed by a client and a server. Typically, the client object invokes a method from the server object that processes the request and sends the result back to the client.

To handle with these distributed objects, there are some technologies, like the Common Object Request Broker Architecture (CORBA) **[2]** from the Object Management Group (OMG), the Component Object Model (COM) **[4]** from Microsoft Corporation, and the JAVA Remote Method Invocation (RMI) **[43]** from Sun Microsystems. These three will be focused in this overview. Currently, Web services are considered by some as an emerging technology for distributed objects over the web. We will focus on these later.

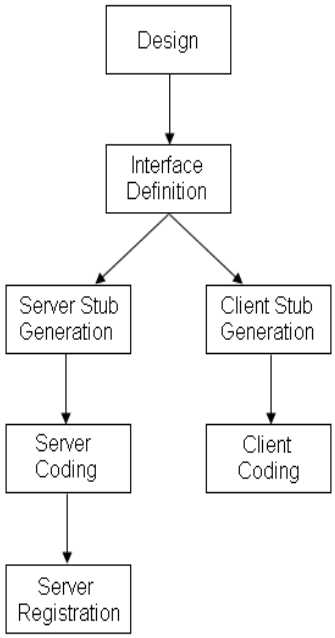

Figure 3: Design and Implementation Process for Distributed Objects [taken from [5]]

Figure 3 shows the steps required to design and implement a distributed objects solution. The first step consists in designing the server object taking into account the kind of client object that it will serve. Then, the Interface Definition concerning that

server object design must be created. From this interface, client and server stubs will be generated. Respecting the generated server stub, the server programmer implements the object described by the interface. This object will then be registered in a directory or name-server, so the client can discover and instantiate the it. The client stubs will be used in the client implementation. This diagram is the base for the model generally adopted by distributed objects technologies, each with its slight variations.

2.1.1. CORBA

The OMG is an international non-profitable consortium in activity since 1989. This group provides open membership of various types (Contributing, Domain, Platform, Influencing, Government, Trial, Analyst, University) and in the present has already several members. OMG provides some specifications to allow the production of applications CORBA-compliant. The success of CORBA came from the idea of heterogeneity and interoperability between distributed objects once it provides support for it through the Object Request Broker (ORB) which provides basic communication mechanisms for distributed and heterogeneous objects.

The first CORBA specification was released in October of 1991. It included the CORBA Object model, the Interface Definition Language (IDL), and the core set of application programming interfaces (API's) for dynamic request management and invocation (DII) and the Interface Repository. Later, OMG started to complete the CORBA specification, with the introduction of CORBA*services* specification which provides several features such as naming, locating objects, or controlling concurrent access to objects. OMG consider that these services provide the facility to construct a distributed system. Later, OMG also started the development of some interface specifications but not with the same relevance than the CORBA*services*. They are called CORBA*facilities* and their role is to help in the development of a distributed system.

It is very important to note that the specifications focused above also include language bindings for development of clients and servers in different programming languages.

This turns implementations in several languages possible.

In respect to the meta-object model, the main features are the fact that the objects are considered as remote objects, whatever their location is, and although the fact each object has an unique identifier, it can have multiple references.

As shown in Figure 4, the CORBA architecture includes seven main components. The most important, ORB core, provides the communication between the other components. It forwards the client request transparently to the server object. The Interface Definition Language Compiler generates client stubs and server skeletons, which are the CORBA server stubs. These two components, together with the Dynamic Invocation Interface (DII), execute the marshalling and unmarshalling of request parameters. This last component, the DII, also support the definition of requests at run-time. The CORBA specification always supported several object adapters, but the last one is the Portable Object Adapter (POA), which defines the object activation and deactivation when needed.

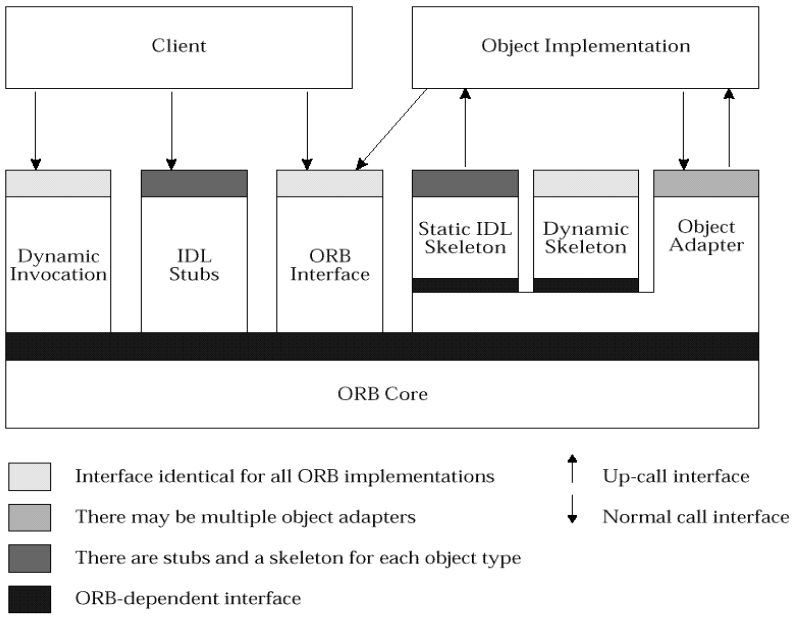

Figure 4: CORBA architecture (taken from [37])

2.1.2. COM

COM is a middleware technology developed by Microsoft in 1993. Like CORBA, COM enables the communication between processes but also supports the creation of objects in some programming languages, which are Visual Basic, Visual C++ and Visual J++. Since inheritance is a feature of the object-oriented paradigm, is important to refer that COM does not include it. This is however overcame with the fact that COM objects can have multiple different interfaces. COM includes some technologies like ActiveX and the Object Linking & Embedding (OLE).

Two main features of COM are the binary encapsulation and compatibility support. The first one means that the coding representation of the server objects can evolve independently from its clients, and in the case of any change in the server code it is only necessary to compile the client code again. The second one stands for the fact that is possible to use the binary code of the implementation of an object in a certain programming environment, while developing in another environment.

As this topic refers to distributed objects, it is fundamental to refer that COM does not support communication between components spread over a net. Because of this, Microsoft introduced an extension to COM, the Distributed Component Object Model (DCOM). This was possible, because Microsoft started to use the Distributed Computing Environment/Remote Procedure Calls (DCE/RPC), which is a system that provides the facility of the software running in computers distributed over a network, as if it were all running on the same machine. Later, an extension to the COM was released by Microsoft, the COM+. The meta-object model of COM has three main entities, classes, implementations and interfaces, and it is important to note that COM interfaces have a Universally Unique Identifier (UUID).

The architecture of the DCOM system, in figure 5, is composed of three layers: application, presentation and session. In the first one, the client has a pointer to an interface proxy and the server has the implementation of the object and a COM class.

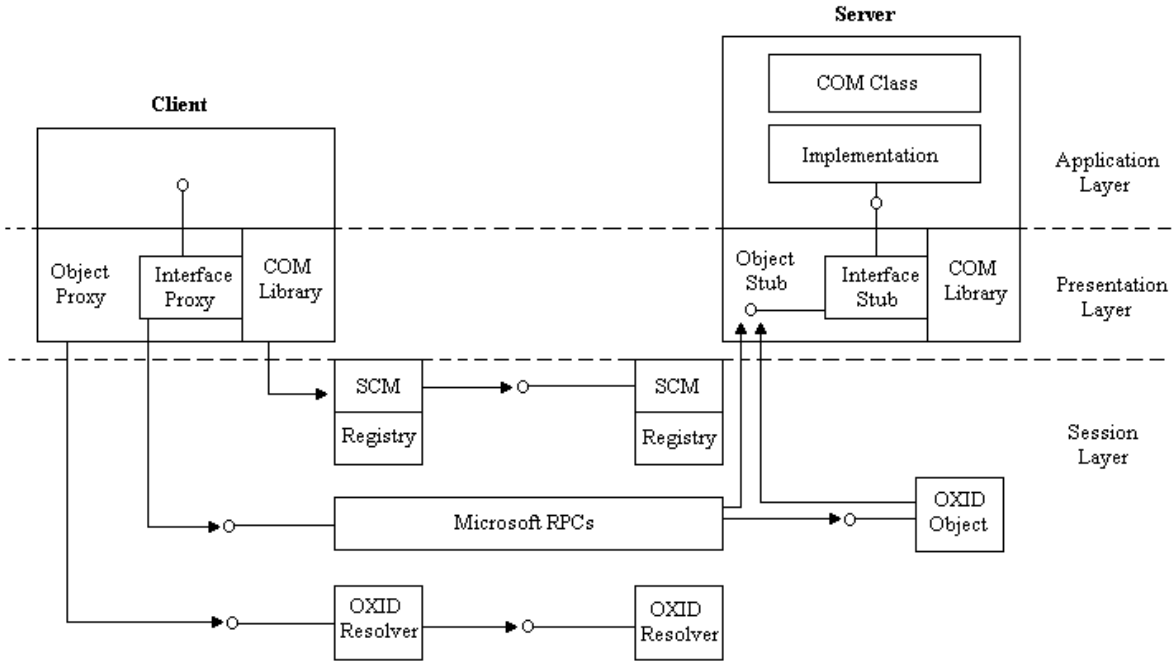

Figure 5: Architecture of DCOM (taken from [5])

In presentation layer, the client and server hosts have only one component in common, the COM library. Moreover, the client host has an object proxy and an interface proxy, while the server has an object and an interface stub. The aim of the object proxy is to marshall and unmarshall pointers and to create an interface proxy for the particular interface through which the request is sent. This interface proxy will then marshall or unmarshall the parameters of all the operations contained in the interface with which it is associated. The object stub in the server host marshalls the object references and creates the interface stub. Each stub is capable of marshalling and unmarshalling the operation parameters for the interfaces it is associated with and of calling the requested operation. These components of the presentation layer, except the COM libraries, are generated by the Microsoft Interface Definition Language (MIDL) compiler. The activation and deactivation of remote objects is controlled by the Service Control Manager (SCM).

2.1.3. Java RMI

Remote Method Invocation (RMI) is an API that extends the Java object-oriented

programming language. This solution was developed by Sun Microsystems and defines the procedure for invoking a method from a server object hosted in a Java Virtual Machine other than the one running the client.

In this model, there are no heterogeneity issues, since Java RMI does not support client and server objects written in other programming languages than Java. They must be written in Java language.

Java RMI meta-object model does not include an IDL, since Java already includes a distinction between interfaces and classes, but also because, as said before, it does not suffer from heterogeneity issues.

Like CORBA and DCOM, RMI provides a mechanism to generate stubs and skeletons from the interface definition in a fully automatic way. These generated stubs and skeletons perform the marshalling and unmarshalling of the parameters. The RMI runtime architecture (figure 6) is simpler than CORBA's or DCOM's.The remote method invocation is started by the client, when it calls a local method to a stub. Clients obtain the references of the remote objects which are on the server side by means of a registry. The server has previously registered its object references in the registry so that the clients can locate and access them.

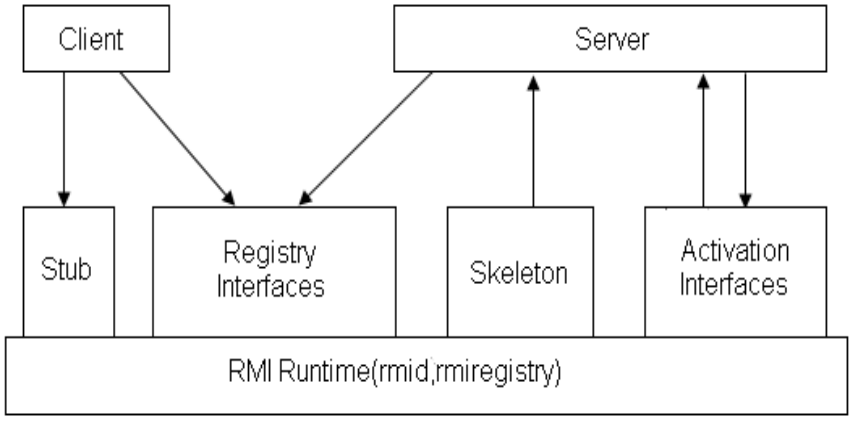

Figure 6: Architecture of Remote Method Invocation (taken from [5])

Server objects can be activated in two different manners. Explicitly by some administrator or implicitly when a remote method invocation is made for that object. In this case, the object has to be able to be activated and use activation interfaces to register itself.

2.2. Service-Oriented Computing (SOC)

SOC **[8]** is an emerging computing paradigm or concept. It consists on the development of a system architecture, a Service Oriented Architecture (SOA), where flexibility, scalability and fault tolerance are the key principles. These principles apply mostly to large distributed systems, since these systems usually suffer from the issue of heterogeneity.

With the growth and demanding of the global markets, enterprises and organizations feel the need of improve their business processes. Therefore, these companies are starting to use SOA since this architecture model brings several benefits in promoting interoperability between their services, and allows a collaborating and cooperative environment between different companies.

Although the term SOA was firstly associated to component-based software technologies such as CORBA or J2EE, Bichier and Lin state in **[6]** state the idea that SOA is now more focused on Web services, and that it has turned into a major research topic.

There are three fundamental technical concepts that support the development of a distributed application using SOA:

Service – A set of features that allow the building business processes in the context of SOA.
Interoperability – Although this term was not introduced by SOA, SOA provides interoperability between the components of a system, and because usually SOA brings heterogeneous systems, it provides the Enterprise Service Bus (ESB) that is an

Information Technology (IT) bus which provides the communication between the different components of the SOA.

Loose Coupling – The principal aim of this concept is to minimize dependencies between the components of a system. This is because there is the need to minimize the impact of modifications on the system and its failures. Loose coupling deals with the requirements of scalability, flexibility and fault tolerance.

2.2.1. Component Architectures

A Service Component Architecture (SCA) is a set of specifications that represent a model compatible with the development of applications based on SOA. These kind of architectures where in the past associated to technologies such as JAVA, DCOM and CORBA. They were in use for several years, but at this moment they are being replaced by the emergent Web services technology. This replacement is mainly due to older technologies limitations, like interoperability issues or availability over the Web caused by the types of connections used to transmit data, which are incompatible with the usual firewall rules. These were some of the motivations for Web services appearance, since this technology seems to solve some of the issues stated above. Despite of the disadvantages referred, the distributed objects technologies present high levels of performance and they still be used by some organizations or enterprises who do not need a Web component in their processes.

Component programming in the Java world is mainly done by resorting two different technologies which are Jini and Java Enterprise Edition (J2EE) through Enterprise Java Beans (EJB). In this last case the J2EE can be seen as the server which allows the use of EJB, a component-based development model. Although Java technology is platform independent, it is not language independent and that can lead to interoperability problems when integrating a large system. Another issue is their application development complexity.

In the case of CORBA and DCOM, the issues of interoperability and complexity do not arise, since they can interact with different programming languages and there are

many developed systems with high-levels of performance. However they do not deliver a good Web support for the reasons explained in chapter 1. Thus, they are not a good choice for those who need to put their business applications over the Internet. From this, at the time of developing a new system,the choice is between the interoperability that Web services provide or the performance obtained with the use of distributed objects technologies.

2.2.2. Web Services

Web services is a technology that appeared a few years ago, and since then has increasingly gain the interest of the community. This interest is mainly due to the fact that Web services provide its operations over the web, without the need to build new applications. The main motivation behind Web services was to build a platform and programming language-independent distributed invocation system out of existing Web standards.

An important thing to note then is that the Web Service concept is not the same as that of SOA or SOC. Although they are often confused, Web services just correspond to a technology that is often used to design and implement SOA. This is because Web services can provide a high level of interoperability between systems. Louridas states in **[7]**, that the idea of inter-operation in a completely transparent manner when running different applications in different platforms and even written in different programming languages motivated the appearance of Web services. This interoperability is possible through the use of the eXtensible Markup Language (XML) standard, which allows the mapping of different programming languages to a widely accepted one. From here, Web services allow inter-operation between heterogeneous systems and business processes.

Compared to the distributed objects technologies, Web services have a loosely coupled approach, i. e., the degree of dependency among the components of a system must be minimized. This is because the integrity of the all system might be compromised by

possible modifications or failures, so their effects must be minimized.

It is also important to state that Web services are not without defects, since they inherit the good and the bad things of the Web, i. e., they are scalable, simple and distributed but on the other hand, they do not support centralized management and are not high performance tailored**.** Moreover they do not handle events, as, for instance, CORBA does. Because of this, Web services are indicated to applications that do not have severe restrictions on reliability and speed.

In the present there are three standardization organizations for Web services:

1. WS-I – Web Services Interoperability Organization[5]
2. W3C – World Wide Web Consortium[6]
3. OASIS – Organization for the Advancement of Structured Information Standards[7]

From this three organizations, only WS-I has the improvement of Web services as a goal. It was founded by companies from the IT world in response to the need of standardization of several specifications. This was in 2002. In the present, WS-I has more than a hundred members. The other two, OASIS and W3C are widely known for the standards they represent.

2.2.2.1. Web Services Standards and Specifications

Web services are based on standards. HyperText Transfer Protocol (HTTP) **[44]** and XML **[29]** are two of them. These are considered Internet protocols and were available before the notion of Web services, although they were an important factor in their acceptance and use.

In the present, there are several Web service standards from different standardization

[5] http://www.ws-i.org

[6] http://www.w3.org/

[7] http://www.oasis-open.org/

organizations, which may lead to interoperability problems. The purpose of the WS-I organization is to solve these issues.

Next, we briefly describe the Web services standards for service description, publication and interaction.

Web Services Description Language (WSDL) **[1]** is recommended by W3C since 2007. WSDL is another XML based language and its role is to describe a Web Service and how it should be used or invoked providing the necessary data to build a SOAP message. In terms of comparison, WSDL is equivalent to the CORBA IDL or to a Java Interface.

UDDI **[24, 28]** stands for Universal Description, Discovery and Integration and is an OASIS standard. It defines a model to publish and discover components of a network, and in this case, of each Web Service. UDDI registry is composed by three components. The White Pages which contain basic information about the providing company like the address or contacts and its known identifiers. The Yellow Pages, which organize the services by industry, service type or geography according to the standard taxonomies and the Green Pages that provide the technical information, such as interfaces and URL location, about how to find and execute a published Web Service.

SOAP **[45]** stood for Simple Object Access Protocol before the release of the version 1.2 which became a W3C recommendation in 2003. In the present, SOAP is not an acronym also because, as referred in **[9],** this protocol has nothing to do with accessing objects. Many think that SOAP is a transport protocol, but this is a wrong idea. SOAP is a XML-based protocol that has the shape of a message which will be sent over a net using a transport mechanism such as HTTP, which is the one mostly used, but there are others like the Simple Mail Transfer Protocol (SMTP) **[46]**. A SOAP message can also be considered as an envelope which can be composed of two elements, the header which contains system information and the body which contains the XML data necessary to the Web Service processes.

Beyond these three standards, there are a lot of specifications associated to Web services. All of them have the prefix "WS-*". As example, there is the WS-Security **[31]**, WS-Coordination **[35]** or WS-TX **[38]**. The aim of these specifications vary a lot, since they can complement each other or even compete. It is not much relevant to focus them in the scope of this work, moreover when they are in different degrees of maturity and acceptance, and are supported by various standards entities.

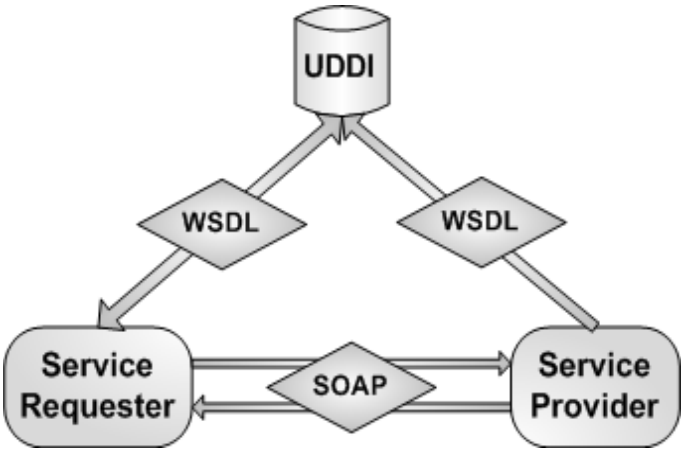

Figure 7: Web Services architecture

Figure 7 portrays a general Web services architecture, including its key elements. Firstly, the Service Provider must create and deploy the Web Service. Then it should publish the respective WSDL file to an UDDI registry. The service broker provides a search service that should be used by a client or service requester to discover a desired service described in a WSDL file. In this case, the UDDI Registry will act as the service broker. After this process, the client can invoke the service from the service provider and the communication will be through SOAP messages supported by a transport protocol like HTTP.

J2EE and .NET are two promising platforms for the development of Web services, but there is not a conclusion of which is better **[41]**. Each of them has advantages and disadvantages. J2EE is a multi-platform and Java-only technology. .NET used to be

only available for Windows platform, but in the present there are solutions, like Mono[8] to provide the development of cross platform applications, once it is an open source implementation of .NET framework.

2.3. Service Composition - Orchestration and Choreography

One of the main features of the service oriented architectures is the possibility of building high-level or complex services and processes. The idea behind this scenario is the integration or composition of multiple services that can be done in two ways, *Orchestration* and *Choreography*. The main difference between these two models is that in orchestration there is the idea of a centralized service that coordinates the other services. The result can be another single service and it can be used further as a basic service or even in other service composition. On other hand, in choreography the principle is peer-to-peer collaboration between services, i. e., choreography defines a model where exists collaboration between different services and it is possible that none of them understands the whole process.

Web Service Business Process Execution Language (WS-BPEL) **[18]** is used to specify business processes behaviour, and it is, at this moment, used to represent the orchestration model, while the Web Service Choreography Description Language (WS-CDL) **[22]** represents the choreography model. The next two topics will describe these two languages respectively.

2.3.1. Business Process Execution Language

WS-BPEL **[18, 19]** is commonly named only BPEL. In the past it was known as BPEL4WS. Currently, it is an OASIS standard, WS-BPEL version 2.0 since April 2007. BPEL appeared from the earlier work of Microsoft with the XML Business Process Language (XLANG) **[20]** and IBM with the Web Services Flow Language (WSFL) **[21]**. BPEL took the best features of these two approaches, adding some other.

[8] http://www.mono-project.org

The main feature of this XML-based programming language is to facilitate the composition of Web services. A composition of Web services consists on the integration of some Services within or across enterprises which allow the creation of higher function and more complex services. These services are then considered business processes. Thus, BPEL may be considered an orchestrating language, because it defines how the services interact or cooperate in order to create a business process.

Processes in BPEL export and import functionality by using Web services interfaces, the WSDL. In the present, BPEL is also considered as a Web services standard, thus it has a relationship with the others standards which were referred in the document previously. As said, BPEL uses the WSDL of a service to know how to invoke it, and because the result of a BPEL process can be also a service, it can be described by another WSDL. To invoke the services BPEL uses SOAP, and UDDI is used do discover and publish services. Thus, BPEL has a tight relationship with the other WS standards. The current version of BPEL supports the following versions of XML specifications: WSDL 1.1, XML Schema 1.0, XPath 1.0 and XSLT 1.0. The current version of WSDL is 2.0, but it is not yet supported by BPEL 2.0, but is possible that a future version support another standards.

Although BPEL syntax is defined in XML language, most of the BPEL process development is made by using BPEL graphical editors, which allow the description of the process as a diagram and then provide the generation of the corresponding BPEL code. In the present, there are several IDE's which provide this tool, like NetBeans, Eclipse or WebSphere among others. This is a great help for one who develop BPEL processes, because the BPEL format might be complex and error-prone.

When defining a business process in a BPEL document, there are four fundamental sections that must be included. Despite this, other elements can be added in a process definition. The first of the four sections is p*artnerLinks,* that include the services that integrate the choreography. The *variables* include the data structures or variables used in the business process. The definition of these variables is made by WSDL message

types and XML Schema Definition (XSD) elements and types. The fault handlers or procedures that should be invoked in case of error are within the *faultHandlers* section. At last, the *sequence* encompasses the definition of the process once it describes the behaviour of the whole process with the activities described in next paragraph.

Other two important elements of BPEL are the basic activities and structures activities. Their role is the creation of the logic of a business process. The basic activities are used for describing the basic steps of the process behaviour, or just to perform a single action. Examples of these are the sending or receiving of messages from services *(invoke, reply and receive)*.

The structured activities define in which order the activities of the process are executed making their composition. They can contain basic or other structured activities. The widely known loops *while, repeatUntil* or *forEach* are examples or structured activities.

2.3.2. WS-CDL (Web Services Choreography Description Language)

Although not directly related with our work, we briefly introduce WS-CDL **[22, 23]**, a novel approach to Web service composition developed by the Web Services Choreography Working Group which belongs to the W3C Architecture Domain. In June 2006 it was released the Web Services Choreography Description Language: Primer, which is a tutorial of how to use the features of WS-CDL specification.

The main focus of the WSCDL, a XML-based language just like BPEL, is to describe the cooperation or collaboration behaviour between different kinds of participants of a business process. The description of this process is defined in an XML document. The behaviour is seen in a general way, in contrast with the centralized view like in orchestration, and that is the major difference between the models. The mechanism behind this scenario is message exchanging. Thus, the result of a choreography model

should be a successful achievement of a goal. Each participant should provide its services and the way they all interact must be in a contract. This contract is nothing more than the description of the choreography. It is important to note that the choreography only includes the interactions of the participating services that may have influence to the choreography's objective or goal.

Unlike BPEL, WS-CDL was developed to allow the composition of services by means of choreography, and then it can not be considered as an executable business process language. It is purpose is to provide collaboration between different types of participants promoting then interoperability because these participants, which are services, can be developed in any programming model and running in any platform.

Several factors motivated the Web Services Choreography Working Group to develop the WS-CDL specification:

- *Re-usability:* a single choreography can be used within several contexts;
- *Cooperation:* the message exchanging between the participating services;
- *Multi-party collaboration:* support for combining already existing choreographies;
- *Modularity:* support for defining a choreography with parts of others choreographies;
- *Exception handling:* handling for errors and exceptions*;*
- *Transactionality:* ability of coordination of the results from the collaboration between the multiple participants;
- *Specification composability:* ability to work together or complementing other WS's specifications.

2.4. Bridging Web Services and other Distributed Object technologies

In this topic we will focus the mapping between Web services and other distributed technologies. We will state the motivations that justify the necessity of this bridging and the solutions that were found by some organizations or enterprises to fill this gap.

The main idea of the bridging process is the invocation from a Web service client of a service from a given distributed object technology in a transparent way, i. e., without noticing that it is invoking a service from a different technology. The client and the server will still use their own standards, but because these standards do not match, a proxy will be used. The goal of this proxy is to translate the messages sent, giving a transparent nature to this process.

2.4.1. Motivation

Bridging Web services and a distributed objects technology simply consists on exposing a service provided by a some technology as a Web Service, e. g., to put a CORBA service over the Web or make it interoperable with a different technology is possible by exposing the service as a Web Service. However this is not the only reason that motivates the use of bridging.

As stated previously, technologies like CORBA, DCOM or RMI used to be the distributed objects market leaders, but lost that position with the porting of services to the Web. So, if Web services are currently more attractive than these three technologies, why just not replace them? This can be answered in different ways, but the truth is that it is very difficult and costly to re-implement existent platforms. That is because an organization or enterprise that has a successful solution developed in distributed objects technology, if wants to replace it with Web services, has to expend a huge investment and it may not be profitable. However, it might be attractive to integrate their developed solution with Web services, and it turns out to be possible by exposing the services as Web services so they could interact. This solution will then protect the investments made in the past. Beyond this reason, it is known that Web services have performance issues, which have been reduced with current implementations, so it is acceptable that one may prefer a distributed object solution. On the other hand, Web services have their benefits, so, at the time of designing a distributed system, the advantages of Web services in respect to interoperability provided and the higher performance that the distributed object technologies provide, have to be considered.

2.4.2. Existing solutions

The state-of-the-art in this area is limited to a few proposals that focus on the idea of bridging CORBA and Jini as Web Services.

2.4.2.1. Web Services and CORBA

By the year of 2000, OMG published was a Request for Proposal (RFP) for CORBA/SOAP Interworking **[10]**. The response for this RFP **[11]**, established OMG's the foundations of what would become WSDL/SOAP to CORBA **[12]** and the CORBA to WSDL/SOAP **[13]** Interworking Specifications, providing a framework for the exposing of CORBA services as Web services. These releases provided the possibility of exposing CORBA as Web services, and drove the development of the concrete solutions.

Elements	5000	10000	50000	100000
Pontifex				
Mean average value	80	164	669	1259
Minimum	62	140	594	1156
Maximum	266	375	891	1484
Standard deviation	22,15	34,11	92,17	81,65
Web Service				
Mean average value	77	150	632	1170
Minimum	62	125	562	1031
Maximum	172	360	828	1360
Standard deviation	13,97	31,79	85,51	97,13
Overhead	3,90%	9,33%	5,85%	7,61%

Table 1: Performance Results (in ms) (taken from [14])

In document **[14]**, a Generic Bridge Generator is described, although it is not available

as solution. It only describes the Java classes that are needed and how to use them. It also refers a performance analysis between a CORBA solution using Pontifex, the Generic Bridge Generator and a pure Web Service. The test consisted in a simple application where a client transmitted a sequence of randomly generated elements of type *long* to the server, which had to calculate the mean average value of those elements. The results are shown in table 1 and as it is demonstrated, the overhead introduced by Pontifex, and bridging in general, is acceptable and that should not undertakes the performance.

It is important to refer that a bridge generator has to translate the CORBA interface definitions to Web services descriptions, i. e., translate the IDL file from CORBA to WSDL and generate a program to provide the mapping of a Web Service invocation to the equivalent in CORBA. Thus, the main functionality of the bridge is to translate the message received from the Web Service to the equivalent CORBA method invocation and then do the reverse operation, that is mapping the result to a Web Service message **[14].**

In tables 2 and 3 the main differences between these two technologies can be found.

CORBA stack	**Web Services stack**
IDL	WSDL
CORBA Services	UDDI
CORBA Stubs/Skeletons	SOAP Message
CDR binary encoding	XML Unicode encoding
GIOP/IIOP	HTTP
TCP/IP	TCP/IP

Table 2: CORBA and Web services technology stacks (taken from [15])

Aspect	CORBA	Web services
Data model	Object model	SOAP message exchange model
Client-Server coupling	Tight	Loose
Location transparency	Object references	URL
Type system	IDL	XML schemas
	static + runtime checks	runtime checks only
Error handling	IDL exception	SOAP fault messages
Serialization	built into the ORB	can be chosen by the user
Parameter passing	by reference	by value (no notion of objects)
	by value (*valuetype*)	
Transfer syntax	CDR used on the wire	XML used on the wire
	binary format	Unicode
State	stateful	stateless
Request semantics	at-most-once	defined by SOAP
Runtime composition	DII	UDDI/WSDL
Registry	Interface Repository	UDDI/WSDL
	Implementation repository	
Service discovery	CORBA naming/trading service	UDDI
	RMI registry	
Language support	any language with an IDL binding	any language
Security	CORBA security service	HTTP/SSL, XML signature
Firewall Traversal	work in progress	uses HTTP port 80
Events	CORBA event service	N/A

Table 3: Comparison between CORBA and Web services (taken from [15])

Apache provides another solution for bridging CORBA services. Apache Axis2 is an

open source Web Service engine developed by the Apache Software Foundation**,** that includes a feature for bridging CORBA services, that is a CORBA module. This module allows a Web Service client to invoke CORBA services in a completely transparent way. The main idea is that the client can invoke a CORBA service without noting that it is a CORBA service. In the present, this solution seems to be the one that best suits for one who wants to expose a CORBA service as a Web Service.

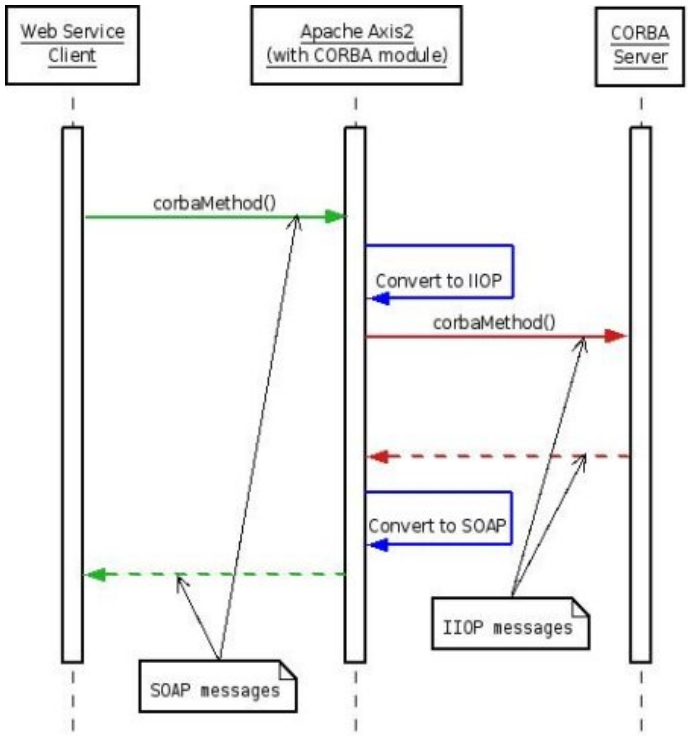

Figure 8: CORBA module from Apache Axis2 (taken from [36])

Figure 8 shows how the CORBA module works. A Web Service Client invokes a method from the CORBA service in a totally transparent way. The CORBA module maps the SOAP message to an IIOP one and then sends the invocation for the CORBA Server, which processes the request and sends the result back to the module. Apache Axis2 does another conversion, but this time from IIOP to SOAP, and then finally sends the response to the Web Service Client.

Three other solutions were found: IONA Orbix **[33]** and Borland Visibroker **[32]**. The enterprises responsible for both products were involved in the response to the RFP

from OMG, but unlike the solutions above, since they are both commercial solutions, they lack on design and implementation information.

2.4.2.2. Web Services and Jini

General brOkering Architecture Layer (GOAL) **[16]** is a service architecture. GOAL was initially thought for the development and management of highly flexible, scalable and self-configurable distributed, and service-oriented applications over the Internet.

This architecture runs on the top of Sun Microsystems' Jini technology, and relies on it for the dynamic registration, service look-up, notification of remote events, distributed object access and platform-independence enabled by Java technology. This lead to the obligation of having service clients and providers implemented in Java, since the communication between services is based on the exchanging of Java-objects.

GOAL itself is not relevant for this work, but is important to refer that later, its authors extended it to promote interoperability between GOAL services and Web services **[17]**.

The interoperability provided by GOAL is of the responsibility of two new added meta-services: the GOAL2WS and the WS2GOAL services.

The GOAL2WS meta-service has the responsibility of generating and deploying a GOAL2WSBridge which makes a Jini service accessible as a WS. A GOAL2WSBridge is at the same time a WS and a Jini client: it receives SOAP calls from a WS client, transforms them in method on the bridged service and then returns the relevant Java results as SOAP objects (Figure 9).

Figure 9: Communication details between a WS-Client and a GOAL-Provider (taken from [17])

The WS2GOAL meta-service generates and deploys a WS2GOALBridge which makes a WS accessible as a Jini service. This WS2GOALBridge is nothing more than a Jini Service which behaves as a WS Client. It receives method calls from a Jini client and returns the SOAP results as Java objects (Figure 10). It is very important to note that each one of the two bridges related is specific to a single service. Each time one wants to make GOAL interoperable with WS's has to create a new bridge. On the other hand, by using these two services briefly described, an already deployed and working service can easily be cast respectively in the WS or GOAL domain.

Figure 10: Communication details between a GOAL-Client and a WS-Provider (taken from [17])

3. The OHMS Platform

The scope of this masters thesis is to develop a platform, that we call **OHMS,** that provides support to create new services through composition, in the context of distributed objects-based platforms.

The architecture of the OHMS platform was designed bearing in mind our desire on a platform that would provide support for the orchestration of distributed objects-based services in the context of their platforms. This lead to another aspect that is having a general solution and not only for a specific technology, i. e., OHMS should accept all kinds of distributed-objects technologies that support a bridging process. Finally, we wanted to avoid at all cost the need for a change in the platform's implementation in order to be suitable for orchestration, i. e., a platform should not have to be reimplemented or redesigned in order to be compatible with OHMS. This property allows the use of OHMS by systems whose code is not public or no longer available (such as legacy systems).

In order to meet our requirements, we designed an architecture composed of two independent modules: the **name-service directory module (**or **directory)** and the **orchestration module.**

This chapter describes the architecture of OHMS and of its composing modules, including the approaches thought to design and implement each module, their inherent issues and the possible solutions.

3.1. Architecture of OHMS

The orchestration of services from distributed-objects platforms will be possible from the moment that a platform registers its name-server in the OHMS platform. This registration allows the platform to choose between the complete or the partial bridging of its composing services.

As referred in the introduction to this chapter, the architecture of OHMS is composed of two modules. The first module, the directory, was implemented in the Java programming language and is the core of OHMS. It is this module that turns possible to bridge services from platforms, by storing information of their name-servers and the logistic to expose their services registry, thus providing platform interoperability.

The second module, the orchestration module, is a platform-oriented extension to the BPEL plug-in from Eclipse. A plug-in created by BPEL Project to provide support for the definition of BPEL processes. Our extension provides a simpler way to access and use BPEL to orchestrate the previously registered and bridged services.

Figure 11 portrays the architecture of the whole platform and the interaction of its modules. These are completely independent and connected through an UDDI Repository embedded in the directory, where the exposed services are published. The directory also embeds a Web Server to provide Web access to those bridged services.

The directory provides a registration peer to accept registrations of both *Technologies*, that includes the bridging logistics, and *Name-Servers,* that includes the information of platforms' name-servers. Both technology and name-server concepts are described in detail in the next section.

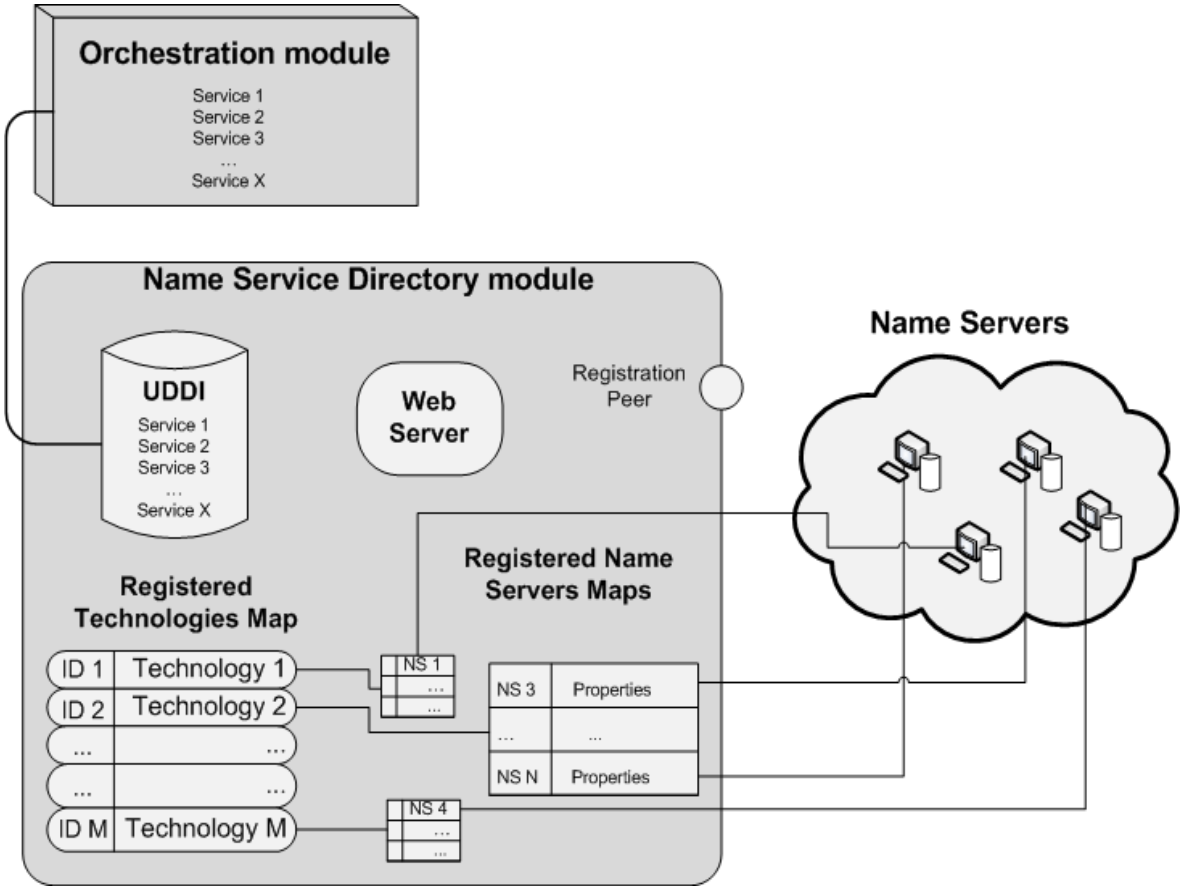

Figure 11: Architecture of OHMS

3.2. The Name Service Directory module

The directory is the core of OHMS. Its general nature turns possible to provide full interoperability in a technology independent way, since it is not attached to a single technology. The directory acts as a server that accepts requests from both technology and name-server clients.

From the directory point of view, a technology is a Java class (or classes) that encapsulates all the logistics necessary to expose a registered service as a Web Service. By logistic we mean the definition of the logic necessary to perform the following three steps:

1. inspect the registry of a technology's name-server, in order to extract the new registered services to bridge;
2. generate the bridge for each of these services; and
3. publish the resulting bridges in the UDDI repository, turning them visible to the orchestration module.

The platform-oriented nature of OHMS leads to the storing the registry of name-servers from the platforms to bridge instead of their services. The registry of those name-servers requires the identifier of an available technology and a Java properties file holding all the name-server specific information. The complexity required to bridge services from a given technology may differ greatly from one technology to another.

In order to make the directory available in a network, our initial idea was to use Web services, but our closer familiarity with CORBA and its respective CORBA naming service droves us to use this technology to rapidly prototype. A Web service implementation is left for future work.

The directory starts and registers itself in a CORBA naming service. Thus, before running the directory's main class, the user has to ensure that there is an ORB running at a specific port, which has to be available to the technologies and name-server clients, in order to be discovered by those clients.

The directory provides to the clients the following CORBA IDL interface to support the *registration, update* and *remove* of both technologies and name-servers. These six operations return an *ErrorCode* that allow the user to know what went wrong, whenever a process fails.

```
module ServiceDirectory{

        typedef sequence<octet> bytecode;
        typedef sequence<classDefinition> classesDefinition;
        typedef sequence<string> techs;

        enum ErrorCode{
            UNKNOWN_ERROR,
            ALREADY_EXISTS,
            NO_EXITS,
            FILE_DOWNLOAD_FAIL,
            NO_ERROR};

        valuetype Technology{
            void run();
            ErrorCode register(bytecode props);
            ErrorCode update(bytecode props);
            ErrorCode remove();};

        struct classDefinition{
            string className;
            string packPath;
           bytecode classCode;};

        interface IRegister{

                ErrorCode registerTech(in string ID,in classesDefinition
                    classesDef, in classDefinition mainClass);
                ErrorCode updateTech(in string ID,in classesDefinition
                    classesDef, in classDefinition mainClass);
                ErrorCode removeTech(in string ID);
                ErrorCode registerNS(in string tech, in string NS_ID,
                    in bytecode properties);
                ErrorCode updateNS(in string tech, in string NS_ID,in
                    bytecode properties);
                ErrorCode removeNS(in string tech, in string NS_ID);
                techs getTechnologies();
                string getWebServerLocation();};
};
```

The interface also defines a *classDefinition struct*. In Java terms, this corresponds to an object and is composed by two strings, *className* with the name of the class file and *packPath* with the path of the package of the corresponding class. The last one, *classCode*, corresponds to an array of bytes that will contain the code of the class. One technology is defined by one or more instances of *classDefinition* and these are the classes needed to implement the bridging logistic we referred previously in this chapter.

The presented interface also provides one more method: *getWebServerLocation*. This operation provides the correct location of the Web Server. The used Web Server has some features available, such as a CORBA module, and this location may be useful for some technology or name-server that will be registered in the directory.

3.2.1 Handling a technology

Technologies are manipulated by *registerTech, updateTech* and *removeTech* methods. In order to register a technology, an user just as to use the *registerTech* method with the three correct parameters: *ID,* the technology identifier; an array of *classDefinition* type, that will include all the classes of the Technology that is registering, except one, that is the main class which will be the third parameter. This distinction is made so the directory is able to know which class should be run to enable a technology. Classes are sent to allow the directory to store them in its own file system, and run the technology when necessary.

At the time of the registration, a new entry will be added to the *registered technologies' map*, an HashMap that associates the received identifier to an instance of the received main class. This identifier should have something to do with the technology it represents, since it will be provided to those who want to register name-servers. Figure 12 demonstrates an incoming technology.

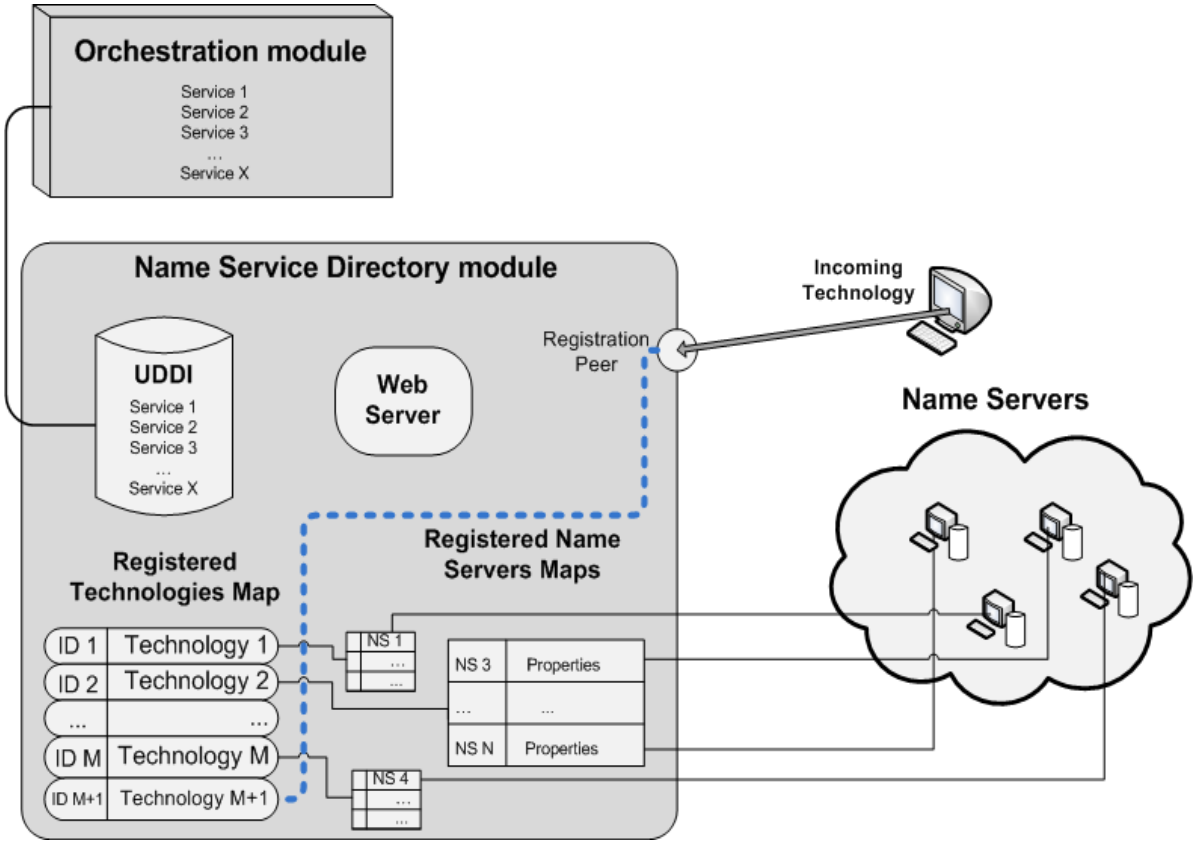

Figure 12: Technology Registration

In order to be compliant with the directory, a technology must be an instance of *valuetype Technology*. This valuetype provides the methods to manipulate incoming name-servers. Although this restriction is not directly imposed by the IDL interface, non-compliance will result in a registration failure.

The updating process, through *updateTech* simply replaces the classes associated to a given technology, causing it to restart

Removing a technology with *removeTech* only requires its identifier in order to discovery which entry to remove from the HashMap.

The way a technology interacts with name-servers registrations is delegated in the technology. However, for each registered technology, the directory keeps a counter

with the number of name-servers associated. When it is zero, technology is stopped, ensuring that only technologies with name-servers associated consume CPU resources.

The directory only supports the bridging of platforms from the technologies currently registered and available, thus providing technology and version independence. The directory accepts registrations from different technologies, such as CORBA or DCOM, and also registrations for the same technology, e. g., “CORBA version 1” and “CORBA version 2”.

3.2.2. Handling a Name-Server

Operations with name-servers are performed using *registerNS, updateNS* and *removeTech* methods. Registering a name-server is done by using the *registerNS* method and includes three parameters: string *tech* that corresponds to the technology that the name-server will be associated; string *NS_ID,* indicating the identifier of the registering name-server, and an array of bytes that includes the Java properties file. That properties file includes information such as the name-server's location or which are the services to bridge.

When a new register of a name-server incomes, the directory relays its registry to the associated technology by invoking method *register* from *valuetype Technology,* with the received properties file as argument.

The process for method *updateNS* is similar to the above described, but this time the *update* method from *valuetype Technology* is invoked.

Concerning method *removeNS,* the directory receives the ID of the technology the name-server is associated and its own identifier. When a name-server is removed, the counter associated to the technology decrements and if it is zero, the technology will stop.

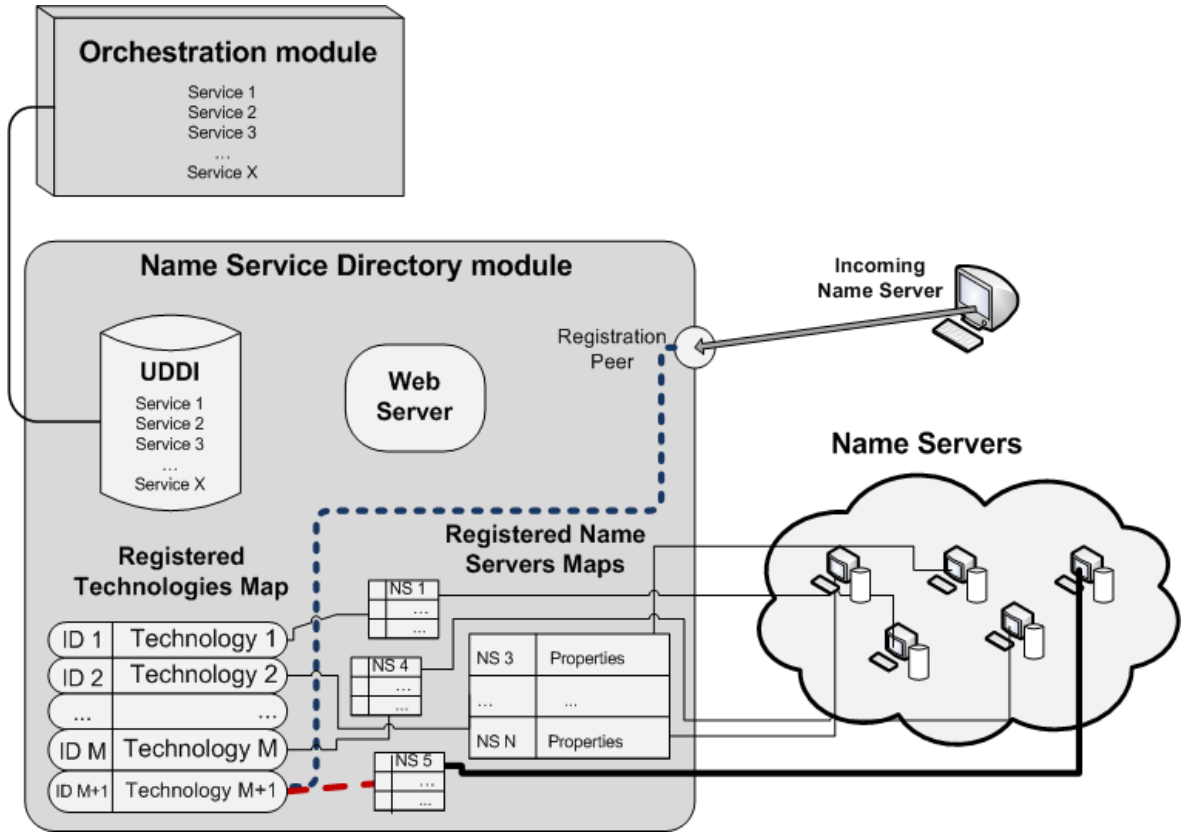

Figure 13: Name-Server Registration

As illustrated in figure 13, the name-server will be associated to the technology chosen, but at the moment of registration of a name-server, the user should ensure that the desired technology is already available. This can be done by using the operation *getTechnologies*. Although this is optional, it is recommended to an user that wants to register a name-server.

3.2.3. Service – bridging and un-bridging:

Once a name-server registers in the directory the bridging begins. The services already registered in a name-server will be bridged if they match any of services included in the properties file. The same is valid for the services that will register in the future.

Note that services are registered directly in the name-server and not in the directory

that only accepts technologies and name-servers. However, as illustrated in figure 14, the service registration triggers an interaction between the name-server and the technology. Technology will discover the newly registered service and will bridge it as a Web service, publishing the on-the-fly generated WSDL interface in the UDDI, thus making it suitable for orchestration. This bridging process depends on the technology implementation.

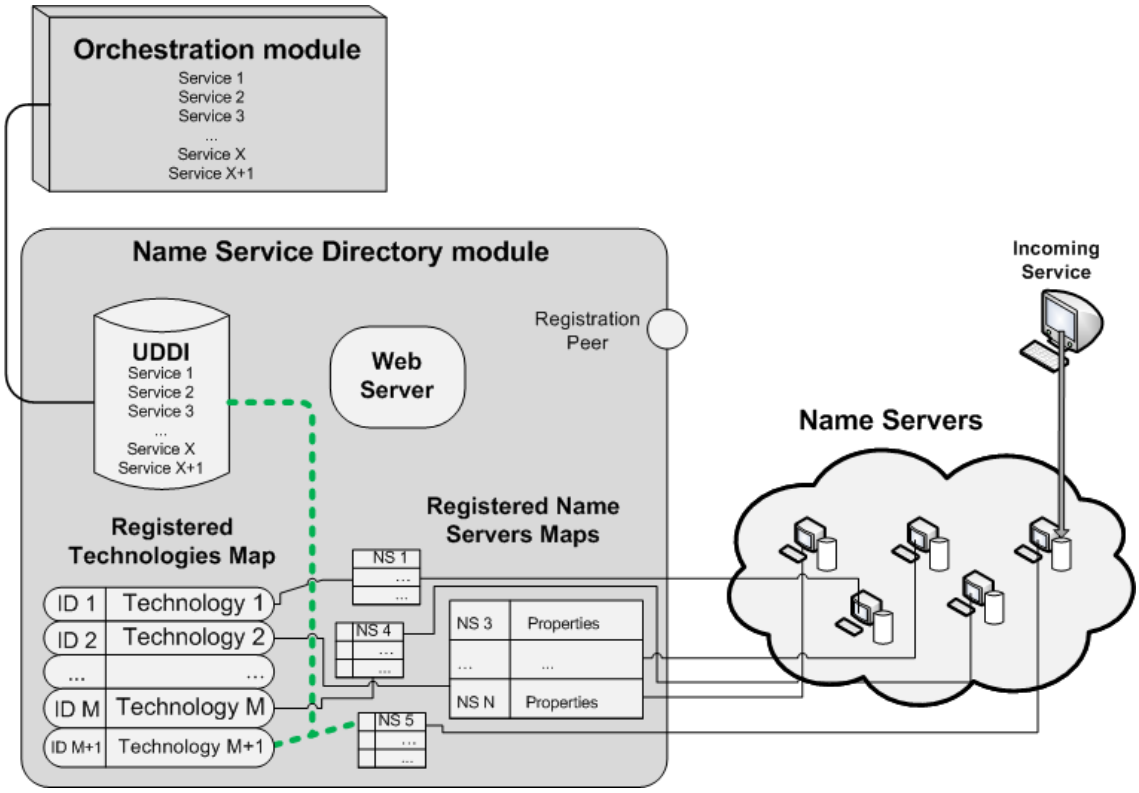

Figure 14: Service Registration

The initialization of the directory starts an embedded instance of Apache Tomcat, version 5.5., developed by the Apache Software Foundation. Tomcat is an open-source servlet container that supports the deployment of web applications. Tomcat is used to deploy the Web Server, Apache Axis2, and the UDDI repository, jUDDI, both from the Apache Software Foundation.

3.3. The Orchestration module

The orchestration module is a platform-oriented extension to the Eclipse BPEL plug-in. It provides support for an user to connect and import services from a distributed-objects platform using the UDDI Repository as intermediate. Once imported, these services can be treated as common partner links in the construction of a BPEL process.

This extension was possible through the use of Eclipse Rich Client Platform (RCP)[9], which is a software that allows the building of Java-based applications on existing platforms or even build those kind of applications from scratch.

The requirement for this module was to show to the user the services available in a given platform. In order to meet that requirement, some solutions were thought, like creating a *pallette* with the available services permanently at sight, creating another view with the same function, or just fill the existing *pallette* with that option. However, the BPEL plug-in has already a strong graphical environment, so we decided not to fill it with more information. The final solution focused on the creation of a new wizard with three wizard pages accessed by a new button.

Currently, jUDDI is the sole implementation available in our extension, mostly because it was already used in the directory module, but for achieving implementation independence, some other UDDI implementations can be inserted and then made available. UDDI interaction is disciplined by an interface, *IUddi.* This provides the support for including any other UDDI implementation. In order to make a new implementation available for users it is necessary to insert that information in the repositories available in the orchestration tool (see last field of figure 17). Thus, it is worthless to create a new implementation of an UDDI repository if it is not include in the available repositories shown in that figure. Moreover, for each new UDDI implementation it is necessary to implement interface *IUddi.* The definition and description of the interface follows:

[9] http://www.eclipse.org/home/categories/rcp.php

```
public interface IUddi {

        Object[] getUDDIInfo(UDDIData data);
        String getWSDL(UDDIData data, String service);
        boolean testUDDI(UDDIData data);
        void setName(String implName);
        String getName();

}
```

- *getUDDIInfo* – Retrieves the services by returning an array of Objects, where each is a service stored in the UDDI repository. This operation is used to present the services in the corresponding wizard (see figure 19).
- *getWSDL* – Returns the WSDL of a specific service. It is used to create new partner links associated to that WSDL interface.
- *TestUDDI* – Test if the UDDI repository is reachable for the plug-in.
- *SetName* and *getName* – Set and get the abstract name of the implementation, for which it is known in the orchestration tool.

In a BPEL process, a partner link is defined by the partner link type that represents the interaction between a BPEL process and the involved parties. These are divided in two categories: the Web services invoked by BPEL processes and BPEL processes invoked by Web clients.

In the scope of our extension to the BPEL plug-in, we created a new button to add new partner links (figure 15). This button, that is next to the *Add Partner Link* button (the cross on the middle), is the one spotted by the arrow. We have chosen to put it right next to the *add Partner Link* button, because as mentioned, it also relies on adding new partner links. Using the middle cross, it is only possible to create an empty partner link and it had to be configured manually, including the association to a Web service through its WSDL.

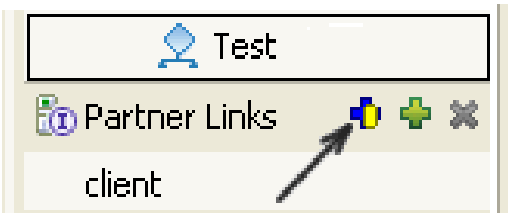

Figure 15: New button: Add a new UDDI Repository

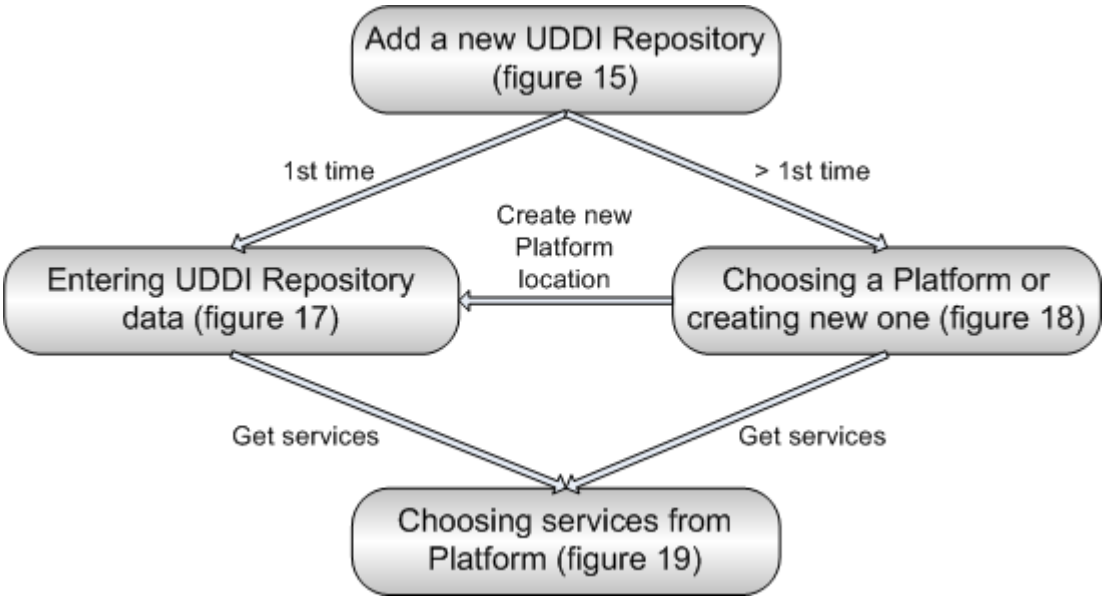

Figure 16: Diagram of wizards interaction

The added button provides support to add new partner links by browsing and selecting the services available from a platform using its UDDI repository as intermediate. This operation requires the selection of the platform to browse, a process that is guided by a wizard, whose behaviour is illustrated on the diagram of figure 16, and is detaily described below.

- First step: Push the created button to browse a platform.;
  - If no UDDI repositories are available, the wizard from figure 17 will be presented. The user can then insert data about a new UDDI repository and add the partner links concerning the services available;

- If UDDI repositories are available, the user will be presented with the wizard illustrated in figure 18 to choose between two options:
  - Use an already available UDDI repository (wizard from figure 19);
  - Insert a new UDDI repository (wizard from figure 17 as described in the first item);
- The addition of a new service is guided by the wizard from figure 21, native to the BPEL plug-in.

The wizard illustrated in figure 17 is the one that allows a user to insert data, that is subsequently stored, about a new UDDI repository. Firstly, it must select the UDDI implementation to use from the ones available. However, at this time, the plug-in only supports jUDDI. After this step, some data about the UDDI is asked. In the referred case, only the field spotted with * symbol is required, but perhaps in others more information will be necessary. After the required information is collected, the user has to push the *Test* button, in order to ensure that the plug-in can acquire a connection to the UDDI repository. A positive response leads the user to follow to the next step, that is adding partner links from the services available in the repository.

Figure 18 presents the wizard where the user can choose between retrieving services from an already available platform or entering data for a new UDDI implementation. User can also remove an available platforms that are no longer necessary.

The choice of creating a new repository results in the presentation of wizard from the previous figure (17) and its associated behaviour.

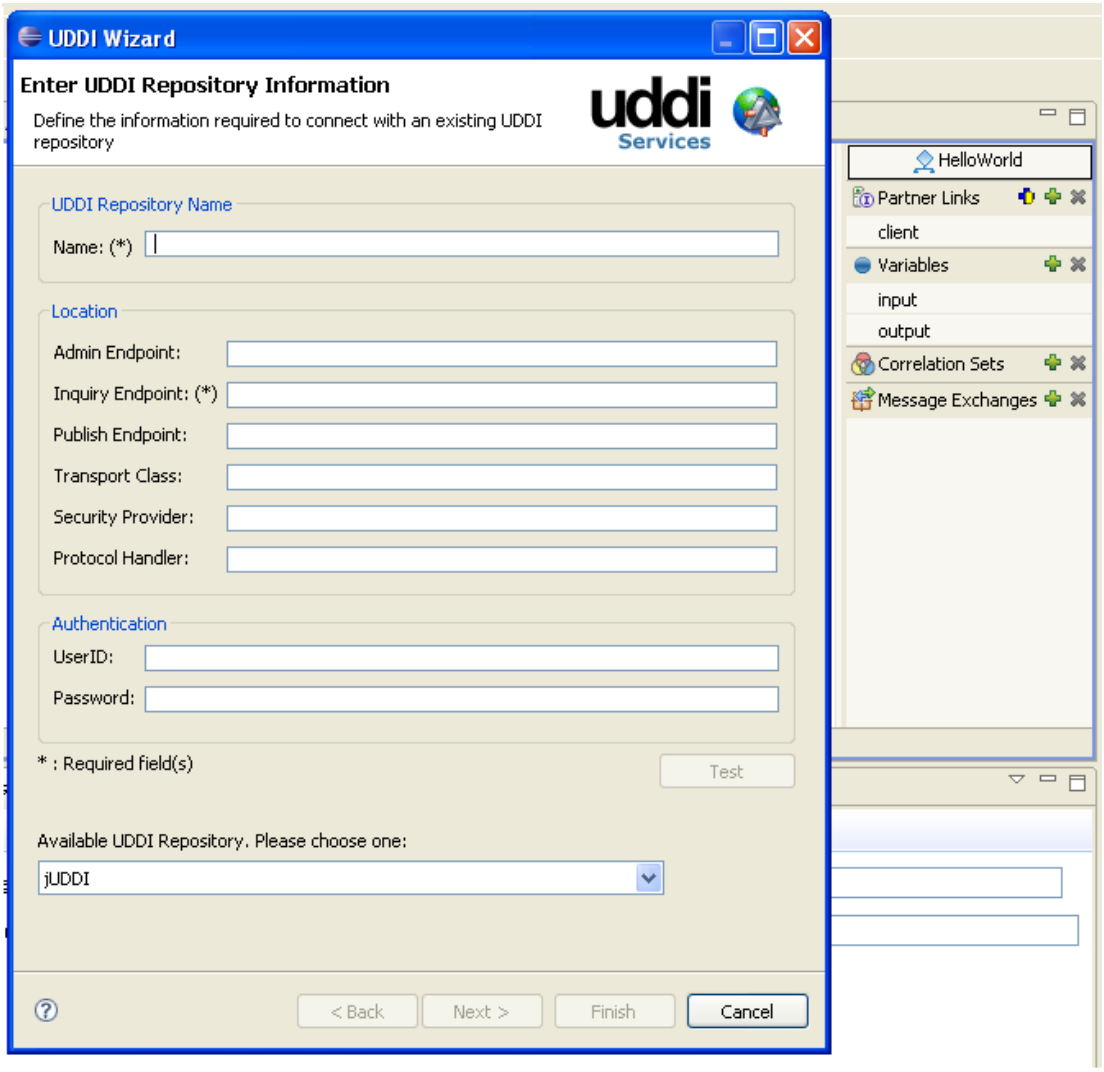

Figure 17: Entering UDDI Repository data

Figure 19 presents the last wizard we created. This is the one that shows the available services from a given platform and that allows the user to add a new partner link associated to a service. Several partner links can be added, but only one at a time because when the user press the mentioned button, will be guided through another wizard, but this time is a native wizard from the BPEL plug-in.

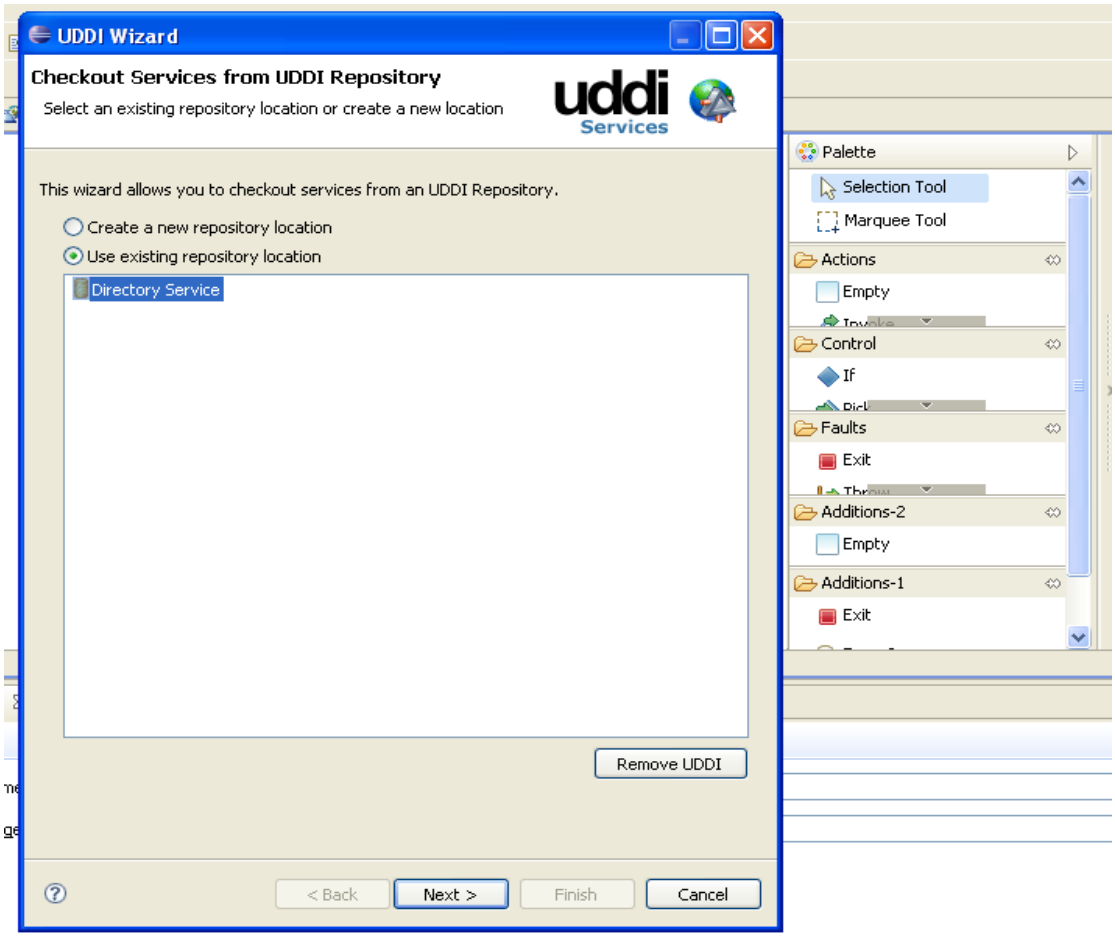

Figure 18: Choosing from existing repository or creating a new repository

Once a service is selected, user must press the *Add as Partner Link* button. Then, a confirmation dialog is presented to the user and the WSDL file respective to the service will be copied to a specified folder, within the BPEL process project file system, and will be always available for the BPEL process. However, if the selected service does not provide a valid WSDL interface, an error window will be shown and user has to select a different service.

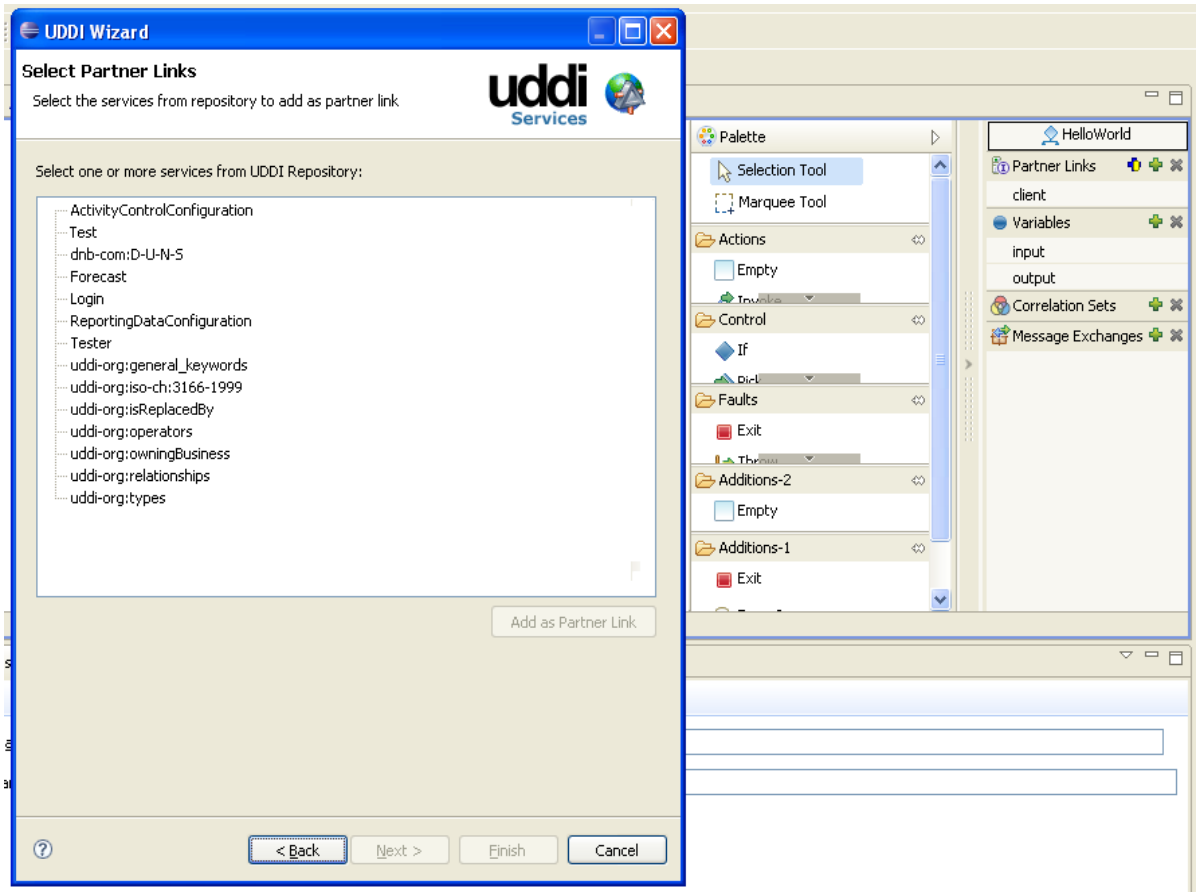

Figure 19: Choosing services from an UDDI Repository

By pressing *OK* in the confirmation dialog, user will be guided through the *Create Partner Link Type* wizard. Although included in the developed tool, the wizard was not developed in the scope of this work. Because of fact, we only illustrate the first page of the wizard in Figure 20. This is the wizard used to configure the type of the partner link that the user needs, making it available to the BPEL process.

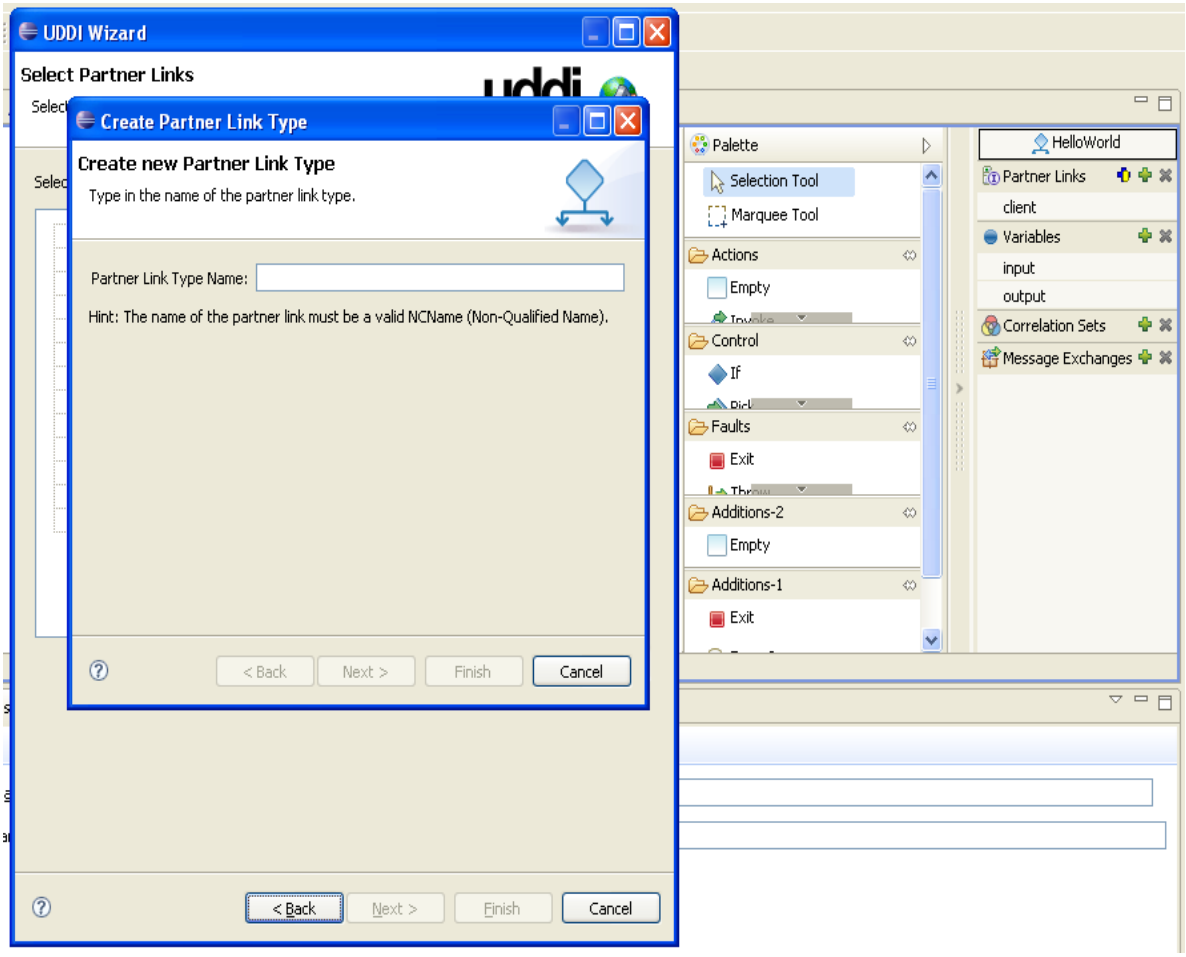

Figure 20: Create Partner Link Type wizard

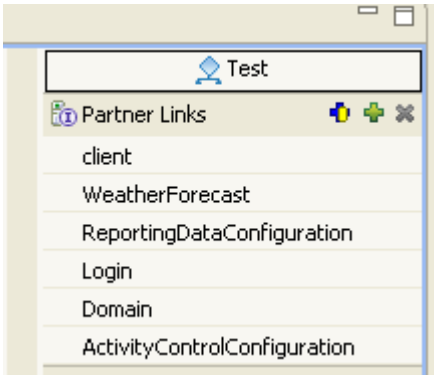

Figure 21: Added partner links

Figure 21 portrays the partner links added, created from services loaded from a platform. This is the final result from the process described in this section. These partner links will enter a BPEL process in the same way as those that correspond to native Web services.

4. On the application to Command and Control Platforms

This chapter describes how OHMS can be applied in a concrete scenario, namely the COMCOP Command and Control platform developed by Critical Software.

We will begin by giving a brief overview of what are Command and Control platforms and on the particular architecture of COMCOP. Those will be followed by how OHMS can be used to expose COMCOP as a Web service-based platform in order to create, by means of orchestration, a couple of new services.

4.1. Command and Control Platforms

Command and control platforms or C2, as they are frequently named, provide the facility of monitoring and controlling assets (either moving or stationary). They used to be associated to military operations, but with the evolution of technologies this situation changed. In the present, this kind of platforms can be assigned to several purposes. One example of this is the Cybercare **[26]**, a command and control system developed to give a quick response in case of medical disasters. This is a concrete example, but these platforms are also used in ground, space or maritime missions.

Command and Control includes all the operations, decisions or actions needed to achieve an objective, and this can be performed in several ways. It can take the format of a regular procedure, like an airplane in the time of landing, or an action which has to be taken in a matter of seconds or milliseconds and has to be executed by a computer, like the controlling of an unmaned air vehicle. In other cases a command and control mission has to be operated by a skilled person, like the management of a

fire fighting team. More precisely, in document **[27]** are the fundamental features that command and control can handle:

- Establishing intent (the goal or objective)
- Determining roles, responsibilities, and relationships
- Establishing rules and constraints (schedules, etc.)
- Monitoring and assessing the situation and progress

With the advance in technology, not everything were benefits, like the higher costs of development and of performing missions, lack of standards for data or several project-specific research that was not really necessary. This promoted the formation of the Consultative Committee for Space Data Systems (CCSDS)[10].

4.1.1. Consultative Committee for Space Data Systems

In the year of 1982, the Consultative Committee for Space Data Systems or CCSDS was formed. As said above, Command and Control started to suffer from lack of standards and interoperability issues, and the major space agencies of the world found the need for the creation of an entity able to tackle them. The result was CCDSDS and it became the proper place for their members to discuss the issues related with development and operation of space data systems with a multi-national forum.

CCSDS develops the *blue books* as their recommended standards. These standards are mostly driven by space mission interoperability and cross support, thus both enabling cross support and reducing the cost of performing space missions.

4.1.2. Command and Control Platform (COMCOP)

COMCOP is a Command and Control platform suitable for aerospace, defence and civil markets. developed by Critical Software in 2006. **[**39**]**

[10] http://www.ccsds.org

Currently, this platform follows a service-oriented architecture and is compliant to one of the CCSDS standards, the Spacecraft Monitoring & Control (SM&C) **[42]**. Next, we will focus the architecture of COMCOP.

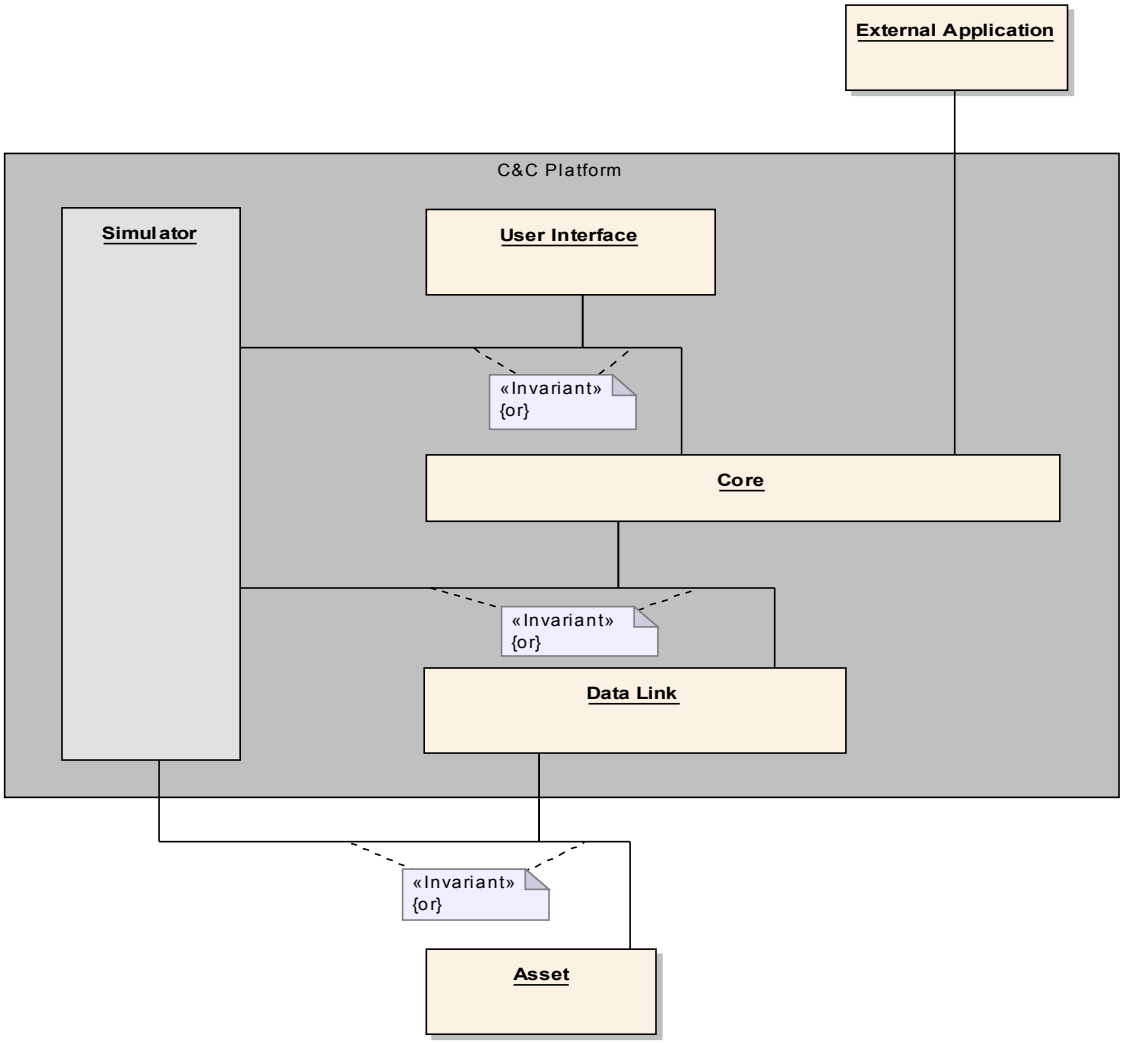

Figure 22: Architecture of the C&C Platform (taken from Critical's Reference Architecture)

In the figure 22, it is possible to see the architecture of the Command and Control Platform and the corresponding layers. Starting with the Asset that is the element that will be monitored and controlled by the Core which in turn processes the requests sent from the User Interface and the data provided by the Data Link. The User Interface is

responsible for the interaction between the Platform User and the Core providing monitoring and controlling displays. The Data Link is a layer that will be integrated with the core through a tailored adapter (Wrapper). The Data Link is represented by any protocol implemented by the Assets being managed.

The Simulator has the facility of simulating the behaviour of the Assets, Data Link or even the entire Core. An External Application is seen as an extension of the Core. This allows the registry of new services that become available to other external or User Interface applications, and the use of the Core Services to provide new capabilities to the system.

A more detailed description of the core services follow, since these will be the subjects

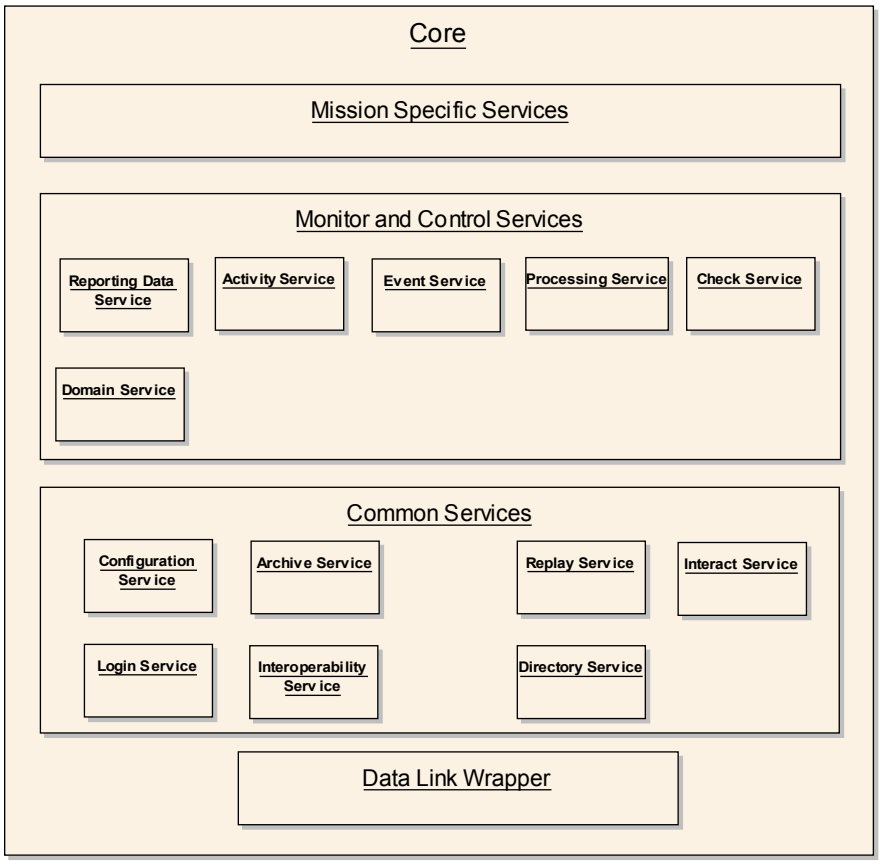

Figure 23: High level concept of the Command and Control Core (taken from Critical's Reference Architecture)

of the bridging and orchestration processes.

Figure 23 represents a high level concept of the *Command and Control Core*. The Core has three layers that can interact with each other. Besides these three layers, there is the Data Link Wrapper whose objective is to map the messages received, from both the Data Link and Core, to messages the receiver can interpret. The three layers include the *Mission Specific Services*, *Monitor and Control Services* and *Common Services*. The services from Mission Specific layer, include those that extend the Core for a specific application or mission. The Services from *Monitor and Control* provide operations for generic monitoring and control of a remote asset. At last, the *Common Services* provide a standard model for services to extend, allowing the specification of standard operations that further simplify the specification of mission services. It provides infrastructure services to support the mission operation services.

4.2. OHMS Platform and COMCOP

This section describes how OHMS can be used to expose a CORBA-based platform as Web-services, and applies it to the concrete context of the COMCOP, populating its Mission Specific Services layer with new services resultant from orchestration.

To meet that requirement, we will have to expose COMCOP services as Web services, creating a bridging logistic for CORBA and registering COMCOP's name-server in the directory module, and then orchestrate those bridged services in the orchestration module by means of BPEL processes.

4.2.1. Bridging CORBA Platforms

In the point of view of OHMS, a technology consists in a bridging logistic that follows the three step process indicated in the section 3.2 of this document and is mapped on the three subsections of this section. Since COMCOP is built on top of CORBA, we

had to create a bridging logistic for this technology.

4.2.1.1. Inspecting the registry of a CORBA name-server

This step consists in discovering new registered CORBA services. On a first approach we thought that the better solution was to poll the name-server to obtain the already registered services and, from that point, intercept any new registration. But to put this feature working, we would have to use the Portable Interceptors mechanism **[40]** from OMG. However, this mechanism is not included in all the available CORBA implementations, and we wanted a portable solution that would allow the use of many, if not all, CORBA naming services. For instance, omniNames**,** the CORBA naming service used by COMCOP does not provide this feature **[30]**.

In order to achieve the required portability, we chose to resort to a polling method, that consists in a task that will connect to the name-server in an interval of time that is determined by the registering name-server in its properties file. This might not be a consensual solution, because this technique usually introduces overhead. However, the interval of time can be decreased to zero, updating the properties file, disabling the connections to the naming service and eliminating the overhead. This may be useful in the case of a platform that do not expect any more service registrations.

The inspection to the CORBA naming service and consequently retrieval of services is done by querying the abstract name bound to the service. However, not all the retrieved services will be bridged. Only those that have a match in the services listed in the properties file.

4.2.1.2. Generating a bridge for each service to expose

To generate a bridge for each single service, we resort to the Apache Axis2 CORBA module. This module requires two files to generate the bridge: an IDL file of the service to bridge and a configuration file. Note that each interface of an IDL file must be associated to a different XML file, so we could have two, three, or even more

configuration files corresponding to just one IDL file. The configuration file consists on a XML file that contains metadata about the service:

- The file name (the new Web Service will be recognized by that name);
- The description of the service;
- The name and path of the IDL file;
- The location of the CORBA service;
- The object name by which was registered in the naming service;
- The interface name, i. e., the modules of the IDL file.

Having both files in the CORBA module directory, Axis2 generates the corresponding WSDL interface for the bridged service, and the proxy required to relay the invocations.

Both the IDL file and the meta-data information, cannot be retrieved directly from the name-server. Although some CORBA implementations provide the *Interface Repository* feature **[58]**, that provides means to obtain information about a service's interface, once again it is not available in all implementations including omniNames **[58]**.

Thus, we decided to follow another approach,closer to the one found in Axis2, that consists in the deposit of both files in a directory within the directory folder tree, that we call repository. With both files available in the repository, the logistic created for CORBA will copy the files, when necessary to the CORBA module folder, and the bridge is generated and the WSDL interface becomes available.

4.2.1.3. Registering the Web service proxy in the UDDI repository

Once the bridging process in concluded the WSDL interface is available in a specific link, e.g., *http://192.168.1.136:8080/axis2/services/Service?wsdl*. The developed implementation will collect that link of the interface, provided by Axis2, in order to use it in the publishing of the recently created Web service in the UDDI repository.

Once this process in concluded, the service is available for service orchestration.

4.2.2. Registering a CORBA Name-Server

For the purpose of registering a CORBA name-server in the directory, we implemented a client, that collects the ID of the technology that will be associated if the registration process is successful. Moreover, the client sends a Java properties file with information regarding the naming service used.

Follows a partial properties file:

```
#Naming service properties

naming-service.id omninames
naming-service.omninames.port 2900
naming-service.omninames.host 192.168.1.316
naming-service.omninames.period 5000

#properties file location

props.dir D:\\CorbaClient\\omniNames.properties

#Services to bridge

services.num 5
services.1 ReportingDataService
services.2 LoginService
services.3 ArchiveService
services.4 ActivityService
services.5 ConfigurationService
```

As is represented above, a properties file contains information about the location of the naming service location, including the host and port that will receive connection, the period for the polling mechanism, the path of the file location, and the services to bridge. Note that this information depends on the necessity of the technology to bridge a service, i. e., a bridging logistic can require more information to bridge services.

Registered name-servers are kept in a map that associate them with their concerning technology.

4.2.3. Bridging and un-bridging CORBA services

The service registration process in the CORBA name service naming service is not altered. The service only needs to know the host and the port where the naming service is running. Thus, the service bridging process will happen every time a new service registration is detected by the bridging logistic developed, in this case, by polling the naming service. After that process, the service is available as Web service and is published in the UDDI repository.

However, the reverse process may also happens, i. e., the service may suffer the un-bridging process, that can be lead by two factors. The first is its removal from the set of services to bridge, in an update of the properties file of the name-server. The second is the detection, from the polling mechanism, that the service has been un-registered from the name-server.

In both cases, the service will be un-bridged, which includes the deletion of the service configuration file from the Axis2 CORBA module folder disabling the bridge. Moreover, the concerning IDL file will also be deleted if it is no longer associated to a configuration file. Without the existence of a bridge, the service registry in the UDDI repository is also removed. However, those files will not be removed from the directory.

4.3. Orchestrating Services in the COMCOP Platform

As we stated before, COMCOP is a Command and Control Platform from Critical Software developed on top of distributed objects, namely CORBA. Also, it follows a service oriented architecture as desired for validating OHMS' ability to expose distributed objects platforms and to orchestrate its services.

The validation scenario consists on populating the Mission Specific Services layer of COMCOP. Although COMCOP architecture is composed by three layers, we consider the other two as its core, since their services are present in every instance of the platform. For that layer populating purpose, we created two simple BPEL processes using the services from the core of COMCOP and one external service.

The first scenario (portrayed in figure 24) is a simple service that creates new activity definitions for a given set of assets of COMCOP. This involves the invoking of the login method from *Login* Service from COMCOP, and, case it is successful, the BPEL process invokes another operation, *addActivityArgumentDefinition,* but at this time from Service *ActivityControlConfiguration* Service. After this is concluded, the returning value, that is an error code, will be assigned to the *addReportingDataDefintion* operation *ReportingDataConfiguration* Service, in order to the result from creating new activities is reported.

The second validation scenario, presented in figure 25, is a little more complex than the first one. Its purpose is to make the choice of the domain assets for a given mission depend on the weather forecast, e.g., in the case of a sea rescue mission, if the sea conditions are good, maritime assets such as a high-speed boat may be considered. Otherwise, the choice should probably rely on a air assets, such as an helicopter.

The implemented BPEL process also invokes the *Login* Service, but at this time, after having the result, if it successful, it will invoke an external weather forecast Web service **[34]**. Depending on the result of this invocation, the *getAssetDefs* operation from *Domain* Service will be invoked in both cases illustrated in the respective figure.

This operation returns a list with assets identifiers, that will be used at the time of invocation of the *addActivityDef* operations that consist in assigning an activity to those identifies assets. The activity assigned shall be different regarding the result of the previous operation, that in turn depends on the result of the weather forecast invocation. Once again, the result returned by the *Activity* Service will be assigned to the *addReportingDataDefintion* operation from *ReportingData* Service.

Both BPEL processes were deployed into Apache ODE[11] and successfully tested. Their resultant WSDL definitions can be published in the UDDI Repository associated to COMCOP, thus making the services accessible and orchestrable as the remainder.

Thus, with the deployment of these two services we were able to attest the functionality of the OHMS platform and of the implemented prototype.

[11] http://ode.apache.org/

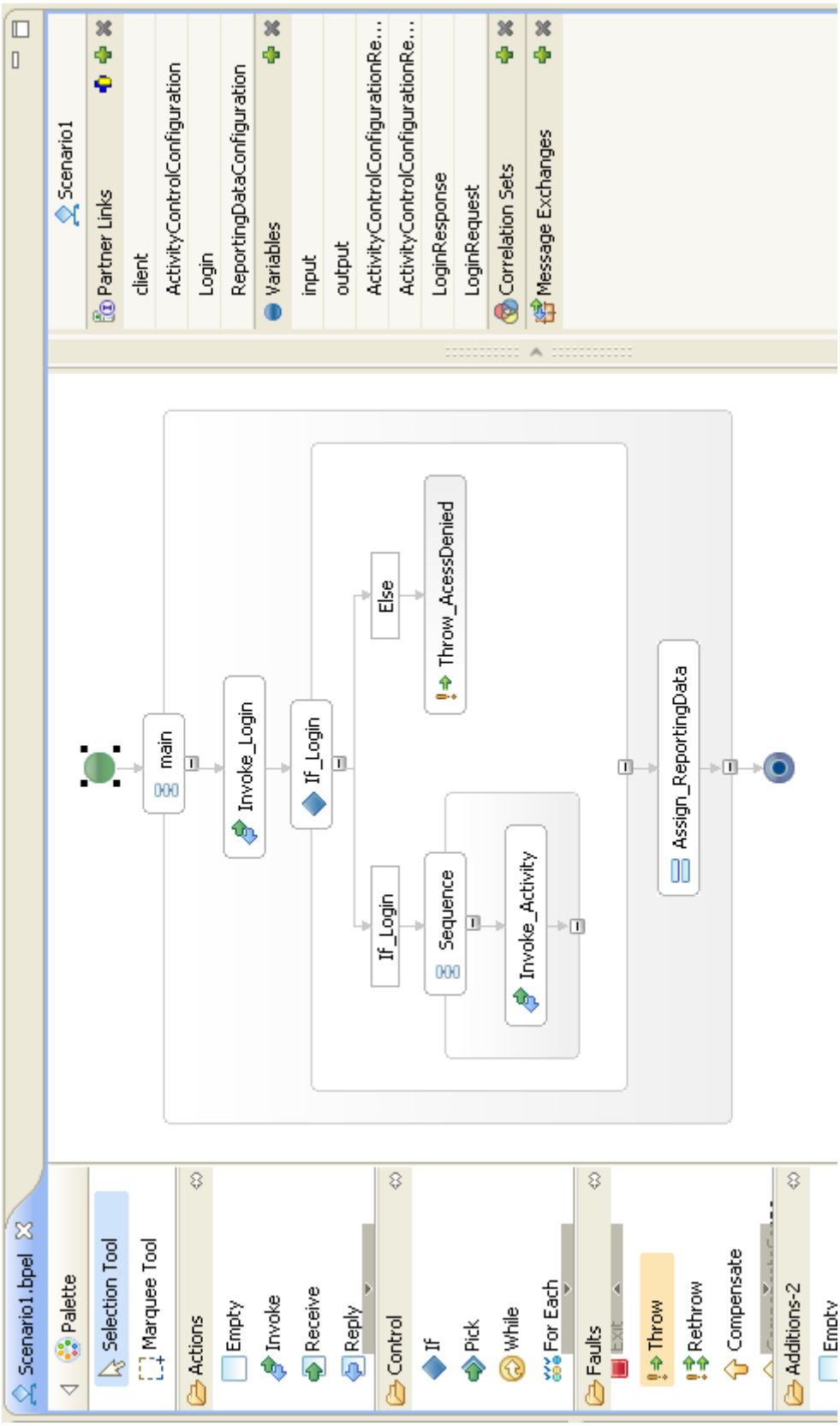

Figure 24: Validation Scenario 1

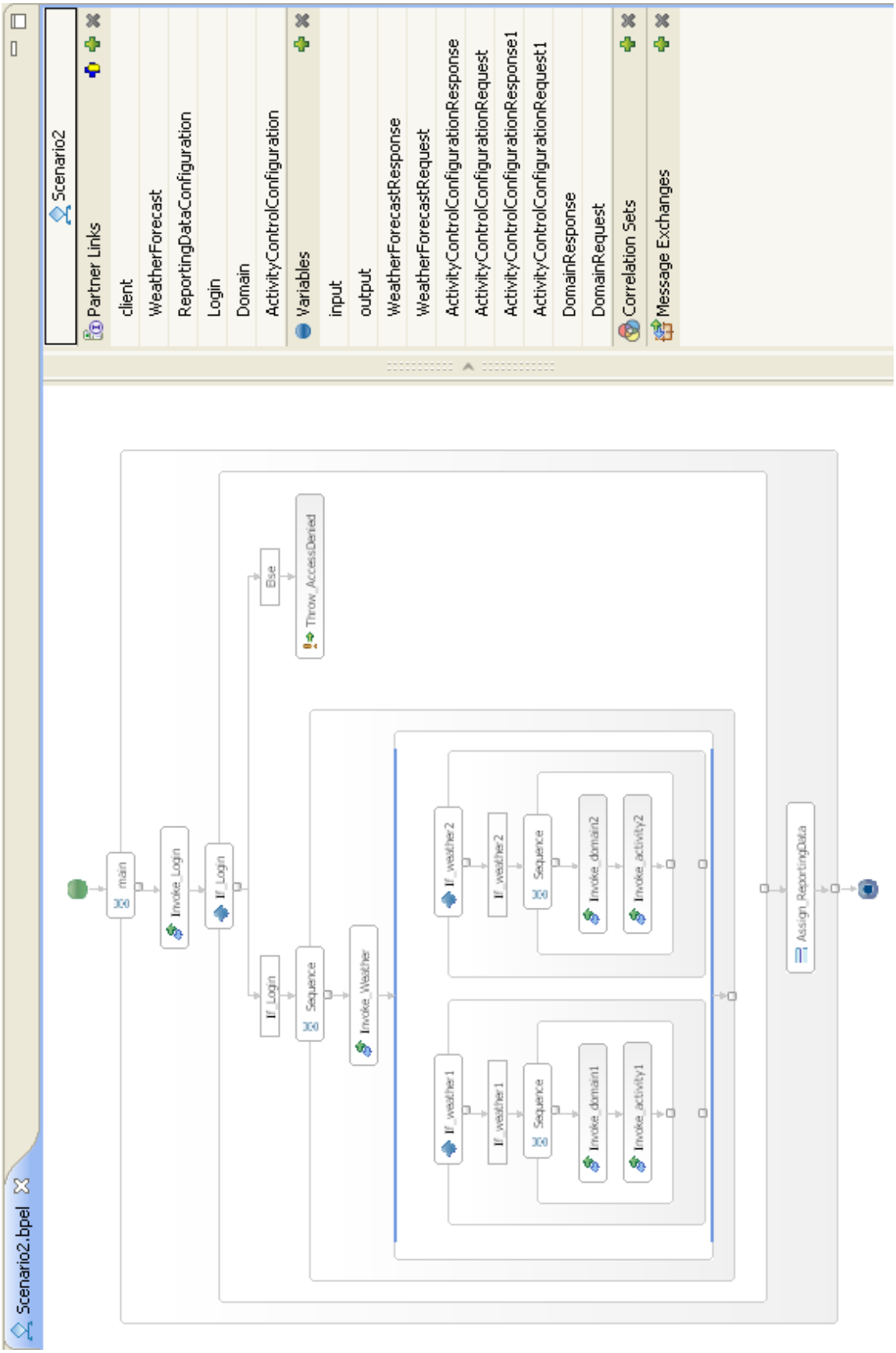

Figure 25: Validation Scenario 2

5. Conclusions and Future Work

5.1. General Considerations

This masters thesis presents OHMS, a platform that provides a simple framework for the orchestration of distributed objects middleware services in the context of their platform.

COMCOP and other distributed objects-based platforms cannot benefit from service orchestration or inter-operate with other platforms. As is evidenced throughout this document, OHMS changes that scenario. By using OHMS, COMCOP and other distributed objects platforms in general are able to profit from service composition, thus creating new services through orchestration. Moreover, we introduced the concept of interoperability. This is particularly important in Command and Control application scenarios, e.g., a disaster scenario, where having different platforms working in conjunction can be vital.

OHMS ports distributed-objects based platforms to the Web services world, enabling Web access and business-to-business interaction, through its both directory and Orchestration modules. It is a technology independent model that allows the registry of new technologies by defining its associated bridging logistic.

By registering their name-server in the OHMS directory module, platforms built on top of distributed objects can automatically expose their set of services as Web services. The implementation design of this module does not require alterations to the original platforms, since the bridging process is completely transparent to the platform. The services to be bridged are specified in a properties file, which is sent to

the directory in the platform's registration process. To prove our concept, we developed a bridging logistic for CORBA technology, that was validated using the COMCOP platform.

Orchestration in OHMS is also platform-oriented. The developed orchestration module enables the retrieval of bridged services from platforms. Thus, this module, that consists in an extension to the Eclipse BPEL plug-in, provides the support for creating BPEL processes using platform services, as well as other services available on the Web. This extension made to the plug-in was possible through Eclipse RCP, known for its steeper learning curve. This fact required a previous contact with that application before starting the development of the extension.

The development of both directory and orchestration modules allowed us to meet the requirements that were initially proposed. Thus, the final result of this work matches our initial expectations.

However, there are some aspects of our work that could be revised, namely the support for new technologies, making OHMS available as a Web service, and improve the solution used for discovering new registered services in our CORBA bridging logistic.

5.2 Future Work

Currently, OHMS only support the CORBA technology. Future work will focus in creating support for others distributed objects technologies. Nonetheless, new distributed objects technologies can be implemented and registered in the directory. The cost involved is almost negligible compared to the porting of a whole platform to the Web service technology. Furthermore, the for registration of both technologies and name-servers can be changed to a Web services implementation. Moreover, the orchestration module can be provided with other UDDI implementations than jUDDI, in order to support a wider range of UDDI repositories. Regarding the OHMS platform, it could be provided with an application for BPEL process deployment and execution.

6. Bibliography

[1] World Wide Web Consortium, *Web Services Description Language,* `http://www.w3.org/TR/wsdl`, 2001

[2] Object Management Group: The Common Object Request Broker: Architecture *and Specification,* Object Management Group, 2001

[3] M. Horstmann, M. Kirtland: *DCOM Architecture,* Microsoft, 1997

[4] Microsoft Corporation, *COM: Component Object Model Technologies*, `http://www.microsoft.com/com/default.mspx`

[5] W. Emmerich, *Engineering Distributed Objects.* John Wiley&Sons, 2000

[6] M. Bichier, K. J. Lin, *Service-oriented computing,* Computer, vol.39, no.3, pp. 99-101, 2006

[7] P. Louridas, *SOAP and Web Services*, IEEE Software, vol.23, no.6, pp. 62-67, 2006

[8] W. T. Tsai, Y. Chen, G. Bitter, D. Miron, *Introduction to Service-Oriented Computing,* Arizona State University

[9] W. Vogels, *Web Services Are Not Distributed Objects*, IEEE Internet Computing, vol.07, no.6, pp. 59-66, 2003

[10] Object Management Group, *CORBA/SOAP RFP, OMG document number orbos/00-09-07*, ftp://ftp.omg.org/pub/docs/orbos/00-09-07.pdf, 2000

[11] Object Management Group, *CORBA Web Services. Initial Joint Submission,* ftp://ftp.omg.org/pub/docs/orbos/01-06-07.pdf, 2001

[12] Object Management Group, *WSDL-SOAP to CORBA Interworking Specification v1.0 formal/04-04-01*, http://www.omg.org/docs/formal/04-04-01.pdf, 2004

[13] Object Management Group, *CORBA to WSDL/SOAP Interworking Version 1.2. OMG Available Specification formal/06-11-01*, http://www.omg.org/docs/formal/06-11-01.pdf, 2006

[14] M. Aleksy, J. Czeranski, M. Schader, *Improving the Interoperability between Web Services and CORBA Using Pontifex - A Generic Bridge Generator*, Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference, pp. 166-166, 2006

[15] A. Gokhale, B. Kumar, A. Sahuguet, *Reinventing the Wheel?CORBA vs Web Services,* In Proceedings of International World Wide Web Conference, 2002

[16] F. Cicirelli, L. Nigro, *A General Brokering Architecture Layer and its Application to Video on-Demand over the Internet,* Informatica, vol. 31, no. 1, pp.29-39, 2007

[17] F. Cicirelli, A. Furfaro, L. Nigro, *Integration and Interoperability between Jini services and Web Services*, Services Computing, 2007. IEEE International Conference, pp.278-285, 2007

[18] OASIS, *Web Services Business Process Execution Language v.2.0, OASIS Standard*, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf,

2007

[19] P. Louridas, *Orchestrating Web Services with BPEL*, Software, IEEE , vol.25, no.2, pp.85-87, 2008

[20] Cover Pages, *XLANG*, http://xml.coverpages.org/xlang.html, 2001

[21] Cover Pages, *WSFL*, http://xml.coverpages.org/wsfl.html, 2001

[22] World Wide Web Consortium, *Web Services Choreography Description Language Version 1.0.*, http://www.w3.org/TR/ws-cdl-10/, 2005

[23] S. Ross-Talbot, T. Fletcher, *Web Services Choreography Description Language: Primer,* World Wide Web Consortium, 2006

[24] UDDI XML.org, *Online community for the Universal Description , Discovery, and Integration OASIS Standard*, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000

[25] World Wide Web Consortium, *Web Services Activity: History,* http://www.w3.org/2002/ws/history.html, 2007

[26] J. Rosen, E. Grigg, J. Lanier, S. McGrath, S. Lillibridge, D. Sargent, C. E. Koop, *The future of command and control for disaster response*, Engineering in Medicine and Biology Magazine, IEEE , vol.21, no.5, pp. 56-68, 2002

[27] D. Alberts, R. Hayes, *Understanding Command and Control,* Monograph, 2006

[28] OASIS, *OASIS UDDI Specification,* http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec, 2005

[29] World Wide Web Consortium, *Extensible Markup Language*, http://www.w3.org/XML/, 2009

[30] OmniORB, *The OMNI Naming Service,* http://omniorb.sourceforge.net/omni41/omniNames.pdf, 2008

[31] *OASIS, OASIS Web Services Security,* http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, 2006

[32] Borland: Borland VisiBroker - A Robust CORBA Environment for Distributed *Processing*, http://www.borland.com/us/products/visibroker/index.html

[33] IONA Technologies: Orbix - CORBA for the Enterprise, http://www.iona.com/products/orbix/welcome.htm

[34] WebServiceX.NET: *USA Weather Forecast,* http://www.webservicex.net/WS/WSDetails.aspx?WSID=68, 2003

[35] *OASIS, Web Services Coordination,* http://docs.oasis-open.org/ws-tx/wscoor/2006/06, 2006

[36] WSO2 OxigenTank, *Exposing CORBA Services as Web Services – Introduction to the Axis2 CORBA Module*, http://wso2.org/library/2807, 2007

[37] The Open Group, *Common Object Request Broker Architecture (CORBA)*, http://www.opengroup.org/branding/prodstds/x98or.htm, 1999

[38] *OASIS, Web Services Transaction,* http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx,

2009

[39] H. Simões, I. Carola, T. Franco et al, C&C-Platform Reference Architecture, White paper, Critical Software, S.A, 2008

[40] Object Management Group, *Interceptors Published Draft,* http://www.omg.org/docs/ptc/01-03-04.pdf, 2004

[41] S. Kachru, E. F. Gehringer: *A comparison of J2EE and .NET as platforms for teaching Web services*, Frontiers in Education, 2004. FIE 2004. 34th Annual, vol. 3, pp. S3B-12-17, 2004

[42] The Consultative Committee for Space Data Systems , *Mission Operations and Information Management Services Area,* http://public.ccsds.org/publications/MOIMS.aspx

[43] Sun Microsystems, *Remote Method Invocation Home*, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[44] World Wide Web Consortium, *HTTP - Hypertext Transfer Protocol,* http://www.w3.org/Protocols/, 2009

[45] World Wide Web Consortium, *SOAP Specifications*, http://www.w3.org/TR/soap/, 2007

[46] Jonathan B. Postel, *Simple Mail Transfer Protocol,* http://tools.ietf.org/html/rfc821, 1982