NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING DEPARTMENT OF TELEMATICS

Implementation of Web services architecture in TAPAS

Master thesis by

Joan J. Vila-Armengol

Trondheim, June 2004

Preface

This report is the result of my master thesis carried out at the Norwegian University of Science and Technology (NTNU) in Trondheim, during the spring of 2004. The master thesis was suggested by my advisor, Mazen Malek Shiaa, and it is related to the Telematics Architecture for Plug-and-Play Systems (TAPAS) research project.

The time spent here in Trondheim as Erasmus student has been very interesting and full of experiences. The topic of the master thesis was challenging and useful for my personal background, Web services is a cutting-edge technology worth of hard work and study. Also, getting to know the TAPAS architecture was a great attempt that provided me with a good knowledge on Java and distributed application programming.

First of all I would like to thank my advisor, Mazen Malek Shiaa, for all the time he has spent explaining and outlining the various aspect of the TAPAS platform, and for advising me on how to go on in the hard moments. I am also thankful for the care and support that my family always has given me, without them it would not have been possible to stay here. I want to acknowledge also my friends here in Trondheim for their support and for all those special moments I had with them. Last, but not least, a big thankful hug to my always supportive and dear girlfriend, Anna.

Trondheim, 19 June 2004

Joan J. Vila-Armengol

Abstract

Telematics Architecture for Plug-and-Play Systems, or TAPAS, is a software architecture that facilitates system installation and management by enabling dynamic instantiation and upgrade of new or existing system components on the network. The TAPAS Basic architecture was developed using Java, thus restricting the interaction of the platform to only Java-based applications. This thesis aims at providing a feasible solution to the application integration problem on the TAPAS core platform. This solution comes by implementing a Web services architecture in TAPAS, and it is demonstrated integrating a XET-based selection engine form the TAPAS Dynamic Configuration architecture.

Chapter one introduces the reader to the application integration issue, Web services basics and outlines the motivation for implementing a Web services architecture in TAPAS. Chapter two presents some related work already done under the TAPAS project. This chapter focuses on the TAPAS Basic architecture and two implemented frameworks under the TAPAS Dynamic Configuration architecture.

Chapters three and four comprise the task of implementing Web services architecture in TAPAS. Thus, adding functionality and application integration support to the TAPAS platform. Chapter three goes through different stages to provide the TAPAS Basic architecture with Web services support. The purpose of this functionality is integrating the XET-based Selection Engine of the Dynamic Configuration architecture. Chapter four describes the implementation details behind the development of the Web services architecture in TAPAS.

A demonstration based on the TeleSchool application for TAPAS is presented in chapter five. This demonstration shows the feasibility of the implementation and how the Selection Engine is successfully integrated with the TAPAS platform. Finally, the last chapter is dedicated to the conclusion and further work.

Table of contents

Table of figures and tables

1 Introduction

In this introductory chapter the importance of adding Web services support to the Telematics Architecture for Plug-and-Play Systems (TAPAS) support platform will be discussed. First, it will be introduced the application integration issue that is the main handicap in nowadays distributed application development strategies. Secondly, Web services technology is presented as the middleware solution supporting the heterogeneous application integration. Some important features relating Web services will also be shown in this section. Then, there is an introduction to TAPAS and the main ideas and concepts behind it. In the last section of this chapter, the solution given to the lack for application integration in the TAPAS core platform will be discussed. Solution that will be achieved by implementing a Web services architecture in TAPAS.

1.1 Application integration

Application integration is one of the most critical issues facing nowadays information technology managers. Application integration is any mechanism that allows different software systems to share, route, or aggregate information, thus improving operational efficiency resulting in reduced costs, and enabling better access to information [1]. So, it is clear that this issue is fundamental to the current model of business process.

In [2], Ballinger focuses on the problem of sharing data, and suggest the Distributed application development as its solution. He defines the Distributed application development as the art and engineering of getting data from one machine to another.

Nearly every application, at some point in its lifecycle, needs to share information and interact with other applications [1]. But, due to generations of proprietary technology, it is not easy to make application systems talk to each another. It is necessary, as a natural evolution of network architectures and business processes, to interact with and integrate application systems that have been built using a variety of hardware platforms and software technology. As said in [1], application integration is hard. There are two main categories that groups application integration strategies: data-level integration and application-level integration.

With data-level integration, applications can share information simply by sharing their databases. This approach has two main problems: it is not suitable for update operations as does not keep consistency at the database and does not let applications share the business logic, the code that implements the business rules. On the other hand, application-level integration takes more time and effort compared to the other approach, but it offers much more versatility and maintains the system consistency. In this approach, an application makes its information and its business logic available to other applications through an Application Programming Interface (API). An API is a programming mechanism that allows an application or system function to expose its capabilities (make them available) to other applications [1]. Using application-level integration is a better strategy when integrating systems, as it is a more complex but cleaner and elegant way of handling the application integration issue. Therefore, this will be the approach used to accomplish with the purpose of this thesis.

Once an interface that exposes the information and the business logic of an application has been created, it is necessary to expose it to the outside world through an open, network-enabled API. This network API is created by means of some communication middleware. This is the point where Web services are introduced; Web services represent a new type of communication middleware. Web services are used to build open, nonproprietary APIs. Web services address two main challenges associated with traditional middleware. The first one relates to pervasiveness and heterogeneity. The second one relates to the cost of ownership associated with integration [1].

Middleware provides easy access to complex system facilities and a consistent interface across various hardware and software environments. Communication middleware is middleware that lets applications talk to one another across the network. It hides the complexity of the network. There are many kinds of communication middleware, each one using its specific protocols. Therefore it is hard to find one middleware package that supports all languages and platforms, i.e., giving a solution to the heterogeneous application integration. Then, Web services appear as a pervasive communication middleware that supports heterogeneous integration, at a low cost and easier to use than traditional middleware solutions.

1.2 Web Services Basics

As said before, Web services addresses the challenges associated with traditional middleware. They provide a common way for an application, written in any language, running on any platform, to talk to any other application. Web services provide a common, platform-independent language that simplifies heterogeneous application integration. Web services represent a new middleware integration platform built on XML and the Web. These are the fundamentals of this technology for which Web services let applications communicate over the Internet. Web services can securely navigate their way through firewall, and they rely on new Internet-based technologies to manage cross-domain security, transactions, and reliability.

In [2], Web services refer to a set of technologies that are the future of distributed computing. Web services handle satisfactory the two largest design goals of most distributed computing solutions; interoperability and loose coupling, understanding for loose coupling that a system exhibits both implementation independence and versioning without breakage. Web services refer to loosely coupled software applications distributed across the Internet [3]. Unlike traditional distributed software applications, for which the distributed components are tightly bound to the application using them, Web services are entirely self-contained and self-describing. As such, a Web service is a full encapsulated, modular unit of application logic that can be found and used by other applications without requiring an intimate knowledge of the inner working of the service. To summarize, Web service exhibits the following defining characteristics:

- A Web service is a Web resource. It is possible to access a Web service using platform-independent and language-neutral Web protocols.
- A Web service provides an interface that can be called from another program. This application-to-application programming interface can be invoked from any type of application. The Web API provides access to the application logic that implements the service.
- A Web service is typically registered and can be located through a Web service registry. A registry enables services consumers to find services that match their needs.

- Web services support loosely coupled connections between systems. Web services communicate by passing XML messages to each other via a Web API, which adds a layer of abstraction to the environment that makes connections flexible and adaptable.

1.3 Introduction to TAPAS

Plug-and-play for telecommunications means that the hardware and software "parts", as well as the complete network elements that constitute a *service system* have the ability to configure themselves when installed into a network (to plug) and then to provide services (to play) according to their own capabilities, the service repertoire and the operating policies of the system [4].

TAPAS (Telecommunication Architecture for Plug-and-Play Systems) [4,5,6] is an active research project at the department of Telematics, which aims at providing a generic platform for enhancing the flexibility, efficiency and simplicity of system installation, deployment, operation, management and maintenance by enabling dynamic configuration of teleservices; network components and network-based service functionality. Since its conception in 1999, a great number of Plug-and-Players, presentations, reports and thesis about the TAPAS architecture, ranging from general architecture overview to specific implemented functionality have been written. Most of these publications are available on the Internet at the TAPAS website at [http://tapas.item.ntnu.no] and they provide a deep knowledge on the whole TAPAS concept and its support platform.

TAPAS as a concept is based on the theatre metaphor. Theatres have repertoires consisting of plays. Plays are performed by actors playing roles which are defined by manuscripts. A director manages plays and the performance of the actors. The concepts used in this analogy are shown in Figure 1-1.

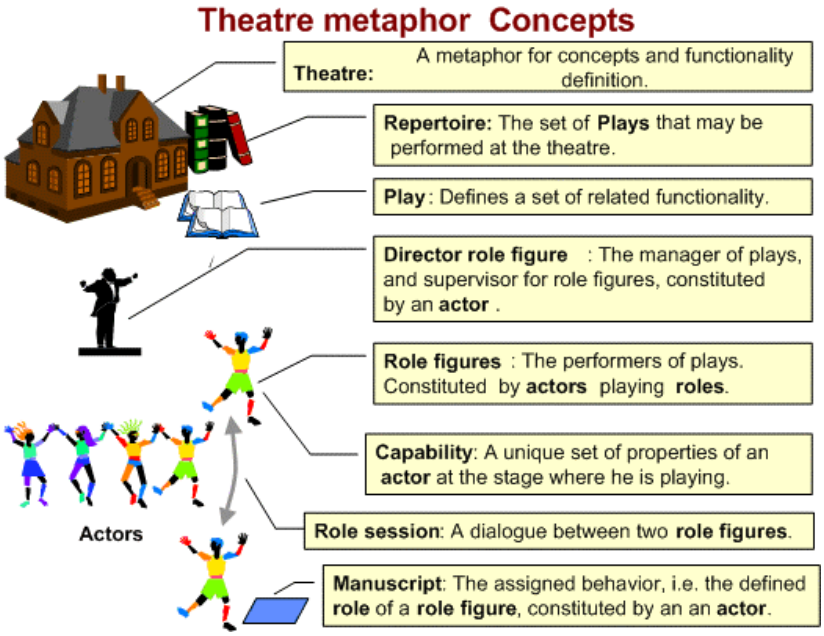


Figure 1-1: Concepts used in the theatre analogy.

TAPAS uses the theater metaphor to define service systems. A service system, in TAPAS, is viewed as a composition of service components. A service component is realized by a role figure based on a role defined by a manuscript and is executed by an actor, which is realized by a software component.

The TAPAS architecture requires a support system for software development, deployment, execution and management. Moreover, the support is also needed for generic user functionality to enable the flexibility features of the system. This support system is denoted as the TAPAS Platform. Four main architectures compose the TAPAS support platform; the TAPAS Basic architecture [5,7]: also called the TAPAS Core Platform is the basis for all dynamic behavior functionality, the TAPAS Mobility architecture [8,9,10]: is the basis for all mobility related issues, TAPAS Dynamic Configuration [11,12]: is the basis for the framework for dynamic configuration and reconfiguration of Plug-and-Play systems, and the TAPAS adaptative Service architecture [12] which handles the complexity and diversity issues of nowadays services as viewed in the global-scale, i.e. availability, capabilities, platforms and technologies, decomposition and distribution of services.

Parts of the specified support functionality have been implemented using Java RMI and socket networking, but this implementation has some lacks that makes it not suitable to support nowadays network heterogeneity and handle the application integration issue introduced in section 1.1. The current TAPAS support platform presents some drawbacks due to its communication infrastructure. The most important issue is the interoperability with non-Java platforms as its execution environment is Java dependant.

In this thesis, a new communication infrastructure for the TAPAS Core Platform is presented and it has been implemented with satisfactory results. This communication infrastructure comprises extended support to the TAPAS Core platform for Web services, providing the core platform with the Web services features and its heterogeneous integration support. The main difference with the original communication infrastructure is the application of an all-web-services node registry and communication model, which is mainly to achieve a totally XML-based architecture.

1.4 TAPAS with Web services support

A TAPAS support system used for experimenting with the concept and functionality of the architecture has already been implemented using Java RMI and satisfactory results have been obtained. However, this implementation lacks the interoperability with other non-Java platforms. The implementation of the communication infrastructure presented here has been built under the support functionality of the TAPAS core platform and it has been implemented using JAVA and Web technologies as a means for service definition, update and discovery.

The thesis is targeting at the application integration issue within the TAPAS platform by means of Web services technology. Two frameworks under the TAPAS Dynamic Configuration architecture have been conducted within the TAPAS project and its core platform: Dynamic Service Management [13] and Dynamic Configuration Management [14]. These two frameworks are based on the same computing mechanism for calculating the (re)configuration plans required when plugging a new service component into a service system. Its core component is a *Selection Engine* consisting of a XET engine [27] that employs the XML syntax and the Equivalent

Transformation Paradigm. Since the XET engine can directly operate and reason about XML expressions and XML clauses, it is employed as the component apparatus to develop an executable engine for reasoning with both dynamic configuration management and dynamic service management frameworks.

The main task performed within the scope of this thesis has been the redesign of the TAPAS communication infrastructure to add extended support for the communication and interaction with the mentioned selection engine. An architecture able to support heterogeneous application integration by means of the Web services technology has been achieved. In the implementation Web services are used for service definition, discovery and XML-data exchange in the communication process. The provided functionality will be shown conducting a case-study that comprises experimentation with an existing TAPAS application, the TeleSchool application [15]. It will validate the developed implementation for integrating the Selection Engine into the TAPAS support platform for a dynamic service management framework.

2 Overview of the TAPAS architecture

A great number of Plug-and-Players, presentations, reports and thesis about the TAPAS architecture, ranging from general architecture overview to specific implemented functionality have been written. Most of these publications are available on the Internet at the TAPAS website at [http://tapas.item.ntnu.no] and they provide a deep knowledge on the whole TAPAS concept and its support platform.

From all the related work on the TAPAS platform, this chapter focuses on four different architectures already implemented within the TAPAS project; the **TAPAS Basic Architecture:** section 2.1 describes the current TAPAS support platform which is the basis for two dynamic frameworks described afterwards, the **TAPAS Dynamic Configuration** is introduced in Section 2.2 while Section 2.3 presents the **Dynamic Service Management** that is an XML-based framework. Finally the **Selection Engine** in Section 2.4 introduces the computing mechanism on which both previous dynamic frameworks are based

2.1 TAPAS Basic Architecture

TAPAS aims at providing a generic platform for enhancing the flexibility, efficiency and simplicity of system installation, deployment, operation, management and maintenance by enabling dynamic configuration of teleservices and network functionality. It allows hardware and software components as well as network elements that constitute a communication system to configure themselves when installed into the network and then to provide services according to their capabilities. Moreover, it permits adaptation of the system to the changing environment, in order to achieve mandated performance levels and at the same time be able to meet user satisfaction.

This section will focus on the Basic Architecture of TAPAS, which defines many of the concepts used in the work presented later in this thesis.

2.1.1 Definition of Plug and Play

Plug-and-play is a concept known from the personal computing area. It means that it is possible to plug-in a component and, without any more effort, the system works

[17]. This type of Plug-and-Play is denoted *Static Plug-and-Play* because both the plugged-in component and the framework has a predefined functionality.

The TAPAS project employs Plug-and-Play in a different and more general way. In TAPAS the component to be plugged-in has some basic capabilities, or external visible properties, that are fixed. The functionality, however, is defined as a part of the plug-in procedure and can be changed dynamically. This means that the definition of individual components, as well as the structure of components, can be changed online. This type of Plug-and-Play is denoted *Dynamic Plug-and-Play*. In addition to making it possible to dynamically change a component's functionality, Dynamic Plug-and-Play is also responsible for making all changes known to possible service users. This way the ability to use the service is propagated.

The difference between Static Plug-and-Play and Dynamic Plug-and-Play can be illustrated by an example. Normally when a cellular phone plugs into a network the system provides Static Plug-and-Play with respect to the telephone service. The functionality of the phone is known in advance. It will work the same way no matter what user is logged on and what network it is attached to, as long as the network can provide the required capabilities. However, one could think of a scenario where a cellular phone that plugs into a network obtains the service it provides depending on its own capabilities, which user that logs on, and which network it attaches to. Here the cellular phone has some basic capabilities, but the functionality depends on the plug-in procedure, and can be changed if for instance a new user logs on.

From now on Plug-and-Play means Dynamic Plug-and-Play.

2.1.2 The Basic Architecture

The TAPAS basic architecture is based on generic actors in the nodes of the network that can download manuscripts defining roles to be played. These nodes are network processing components, such as servers, routers and switches, and user terminals, such as telephones, laptops, PCs, PDAs, etc.

In TAPAS, a service system consists of service components, which are units related to some well-defined functionality defined by a play. A play consists of several actors

playing different roles, each possibly having different requirements on capabilities and status of the executing system. A role-session is a projection of the behaviour of an actor with respect to one of its interacting actors. An actor is a generic object, which will constitute a role figure by behaving according to a manuscript defining the functional behaviour of that particular role in a play. A service component is realised by a role figure based on a role defined by a manuscript. A role figure, however, is realised in an executing environment in a node and is utilising capabilities. A capability is an inherent property of a node. A node may have several capabilities. These capabilities are offered to actors, which constitute role-figures in various plays. The ability to play roles depends on the defined required capability and the matching offered capability in a node where an actor is going to play. Examples of capabilities are processing, storage and communication resources (e.g., CPU, hard disk and transmission channels), standard equipment (e.g., printers and media handling devices), special equipment (e.g., encrypting devices), and data (e.g., user login and access rights) [28].

2.1.3 The TAPAS layered model

TAPAS basic architecture has several layers as viewed from the functionality and support platform point of view. These layers of the TAPAS layered model are shown in Figure 2-1. Then, an operational Plug-and-Play system example is presented according to this model, and a brief description of each layer is given based on the support functionality.

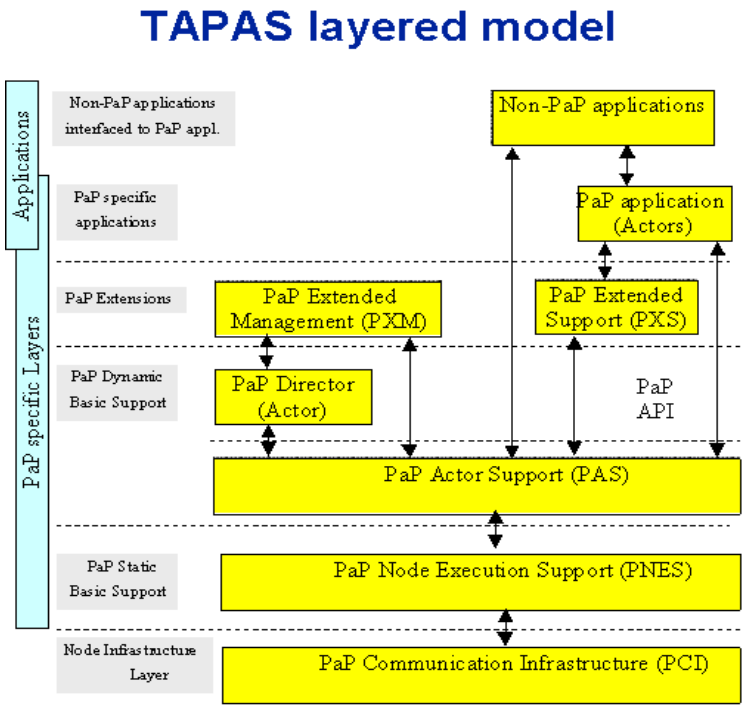


Figure 2-1: The TAPAS layered model.

Figure 2-2 shows an example that illustrates the structure of the support functionality. Then, an overview on the support functionality and the description of the concerning layers is given.

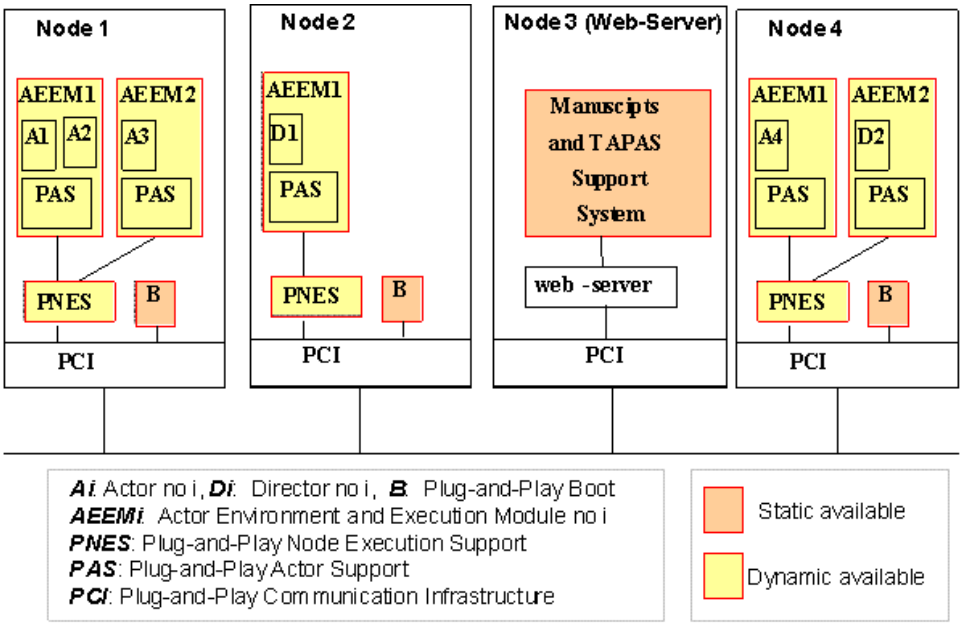


Figure 2-2: Example of the support functionality structure.

The Actor-environment-execution-module (AEEM) is a process or thread that executes a collection of actors with associated Plug-and-Play Actor Support (PAS). A collection of actors is here one or more actors constituting application role-figures or director role-figures. The TAPAS platform basic functions supported are provided by the procedures: PlayPlugIn, PlayChangesPlugIn, PlayPlugOut, ActorPlugIn, ActorPlugOut, ActorBehaviourPlugIn, ActorChangeBehaviour, ActorBehaviourPlugOut, RoleSessionAction, ChangeActorCapabilities and Subscribe.

Plug-and-Play Node Execution Support (PNES) makes it possible to run Plug-and-Play software on a node, and for Plug-and-Play functionality (i.e. executed by actors) on different nodes to interact with each other. PNES is able to receive requests from other PNESes, interpret these requests and take proper actions. PNES will also do start-up and initialization of PASes or PCIs if that is required. PNES implements the Plug-and-Play functionality that is termed the Plug-and-Play Static Basic Support in the model. Static in this sense means that changes/extensions of the PNES functionality must be backward compatible with earlier versions because this functionality represents the "bootstrap" that is necessary to be able to run Plug-and-Play applications. Only this functionality must be manually installed at a node before Plug-and-Play applications can be installed and activated.

Plug-and-Play Actor Support makes it possible to create actors within the context of an operating system process/thread, to give these actors behavior, and to communicate information between these actors and their environments. There will be one PAS instance within each Actor-environment-execution-module as defined above.

Director is both responsible for the management of the Plug-and-Play application definitions, i.e. its part of the repertoire- and manuscript-bases, and for the management of information concerning actors, i.e. its playing-base. A director is involved in many of the functions related to the services provided by PAS.

2.1.4 TAPAS communication model

The TAPAS communication model presented in this section is meant to be a valid solution for the distributed TAPAS architecture. This communication model has three

main features; it is based on a synchronous communication model, it has a specific addressing and routing system, and its registry mechanism. The implementation of Web services architecture in TAPAS comes with changes in the synchronous communication model and in the registry mechanism. So, it is important to know the current communication model to better understand the changes made to it.

2.1.4.1 The Synchronous communication model

Figure 2-3 shows the *Synchronous communication model* using an example with two nodes having different actors under the TAPAS support platform.

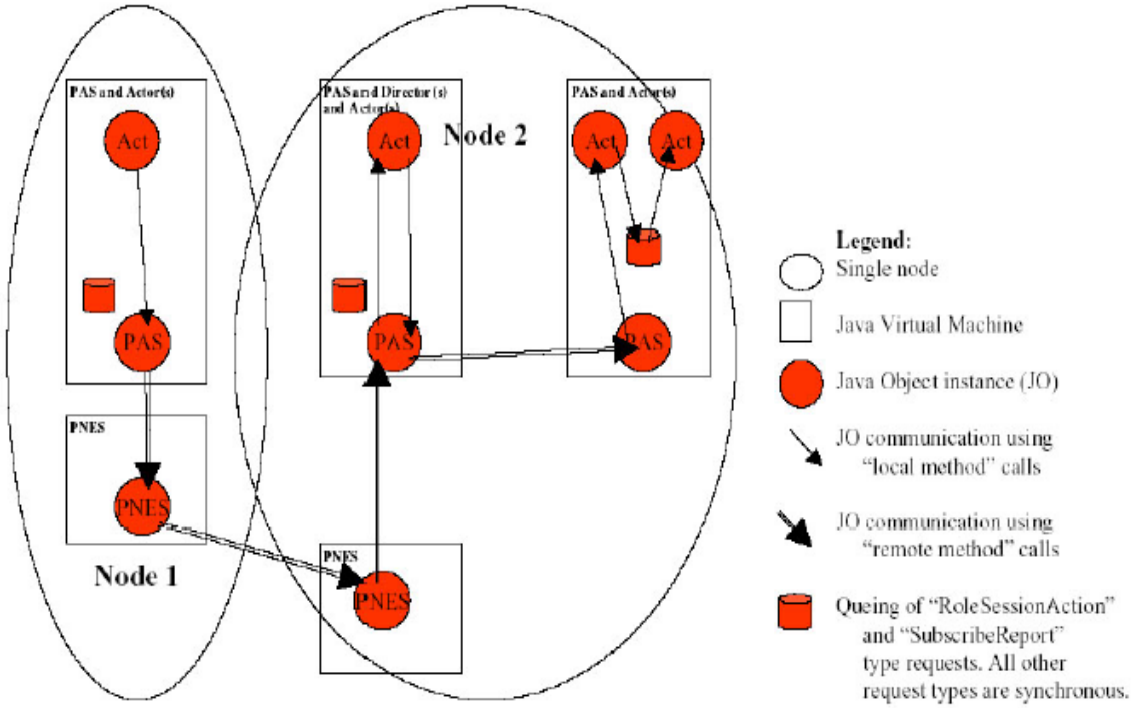


Figure 2-3: The TAPAS Synchronous communication model.

- The Synchronous communication model means that a request from one object to another is executed before the execution continues at the requesting object.
- Actors only communicate directly with its owner PAS by using local method calls. PAS communicates directly with PAS instances located at same node and with the one and only PNES instance at same node, by using remote method calls. PNES instances communicate with PNES instances at other nodes by using remote method calls exact the same way as when PAS communicates with PNES. Using

remote or local method calls are completely hidden from the application point of view.

2.1.4.2 TAPAS specific addressing and routing mechanism

The addressing and routing mechanism of the TAPAS communication model is based on the Global Actor Identifier concept. The same address identifier type, i.e. the Global Actor Identifier (GAI), is used for addressing any kind of addressable entity. The four addressable types are PNES, PAS, Actor and RS (i.e. Role Session). In Figure 2-4 the addressing and routing mechanism is outlined. A GAI value is subdivided into different parts, and where the mandatory part denoting the addressed entity type (i.e. the values PNES, PAS, Actor, RS) determines which of the remainder parts that needs to be used. E.g. a PNES instance is uniquely identified by the PNES identifier value, while a role session is identified by a PNES identifier, a PAS identifier, a Director Actor identifier and a unique number within the specified Director Actor.

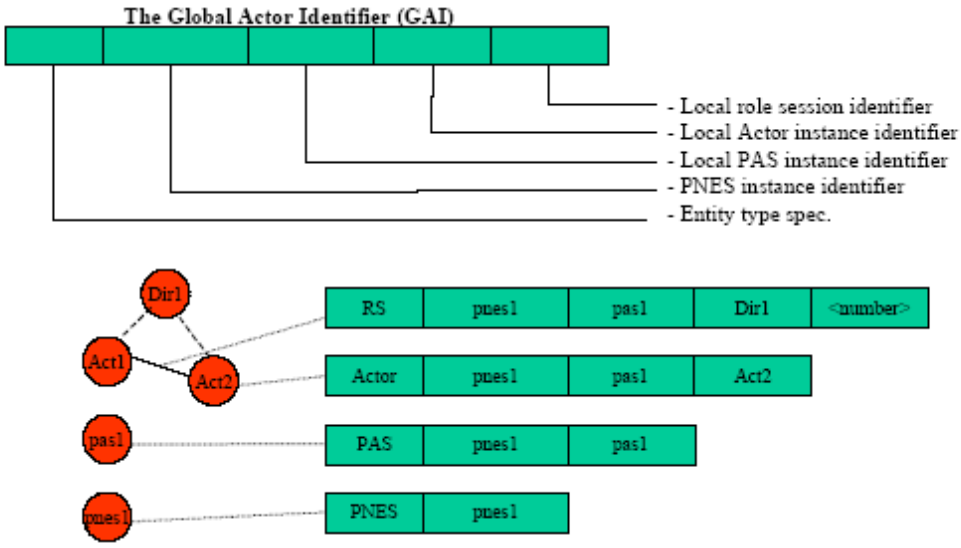


Figure 2-4: Addressing and routing values.

2.1.4.3 The Entity registry mechanism

Some entity registry mechanism must be used in order to find Plug-and-Play entities in the Internet. For this purpose, the Java "rmiregistry" is used to register the PNES and PAS entity types. But if we refer to local actor instances belonging to the same PAS, an internal mechanism based on tables is used. Both registry mechanism are

shown in Figure 2-5. In both situations, between nodes or locally, the GAI identifier for the addressed entity is the key for identifying a specific entity, and the result of an identification is a handle to a Java object representing the addressed entity instance.

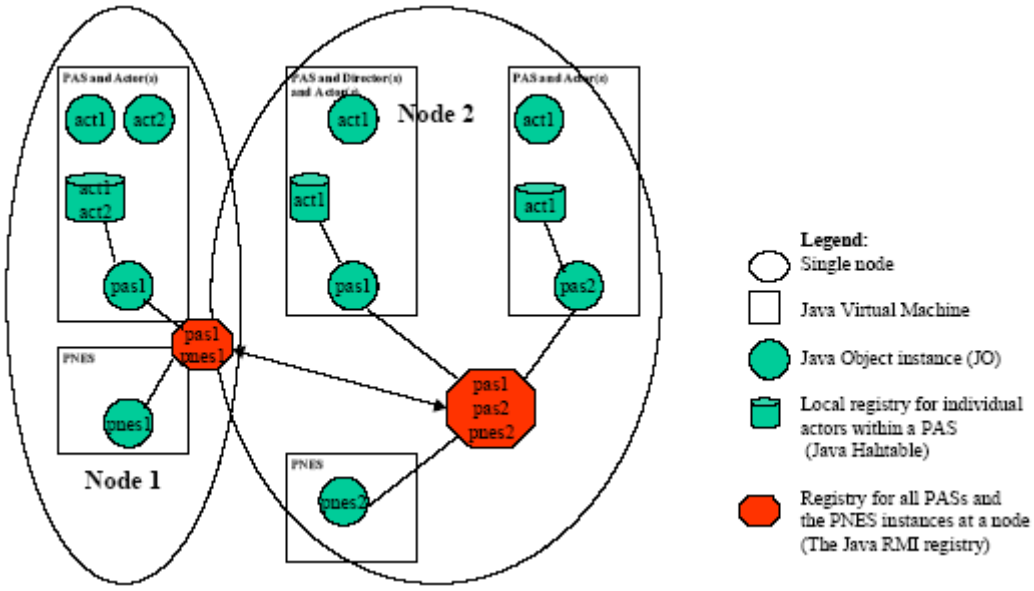


Figure 2-5: The Entity Registry

The figure shows that this mechanism is used by PAS and PNES instances. There will be separate "rmiregistry" "instances" on each node, and the different rmiregistry instances do interactions with each other which it is transparent for the applications.

2.2 TAPAS Dynamic Configuration

The TAPAS dynamic configuration framework is based on the TAPAS basic architecture. The TAPAS Dynamic Configuration architecture, as introduced is section 1.3, is the basis for the framework for dynamic configuration and reconfiguration of Plug-and-Play systems. It provides representation, computation, and reasoning mechanisms for semantic description and matching of capabilities and status information. These concepts on dynamic configuration, i.e. capabilities and status, are introduced in section 2.2.1. These concepts are necessary to understand the framework presented later in section 2.2.2. The Dynamic Configuration framework is a framework already implemented and it is fully described at [14].

2.2.1 Dynamic Configuration concepts

In TAPAS *Capabilities* can be classified into *Resources*, *Functions* and *Data*. *Resources* are considered to be physical hardware components with finite capacity, such as processing, storage and communication units; *Functions* are pure software or combined software/hardware components which perform particular tasks; and *Data* is just data, the interpretation, validity and life span of which depend on the context of the usage.

Besides being classified as resources, functions and data, capabilities can also be characterised by their varieties, i.e., whether they are optionable or absolute. In addition, each capability can be associated with certain operational and behavioural properties.

Status is, at a certain time instant, the situation in a Plug-and-Play system with respect to the actual number of nodes, playing plays, traffic situation as well as operational and non-operational states of each node or each capability component in the system. It can comprise observable counting measures, measures for QoS or calculated predicates related to these measures.

There exist two types of requests: Service Requests and Service Component Requests. A Service Request message request the installation of a Plug-and-Play service not yet installed in the system. A Service Component Request message requests the instantiation of a service component in a running system, i.e. the plug-in of an actor. Each request carries information on which play or which role is requested.

Trouble reports may also be of different types depending on the type of trouble it describes. If the message is created because an actor is experiencing trouble reaching another actor, the trouble report is said to be of type actor unreachable report. However, if the trouble report is sent because an actor finds that the node where it is currently running can not offer sufficient capabilities, an insufficient capability report is sent.

2.2.2 The Framework

The architectural framework for *Dynamic Configuration* in TAPAS is illustrated in Figure 2-6, and comprises the following main entities:

1. *Capability & Status Repository (CSRep)* stores specifications of capabilities offered by components in a system and maintains information reflecting the situation and status of the system at a particular time.

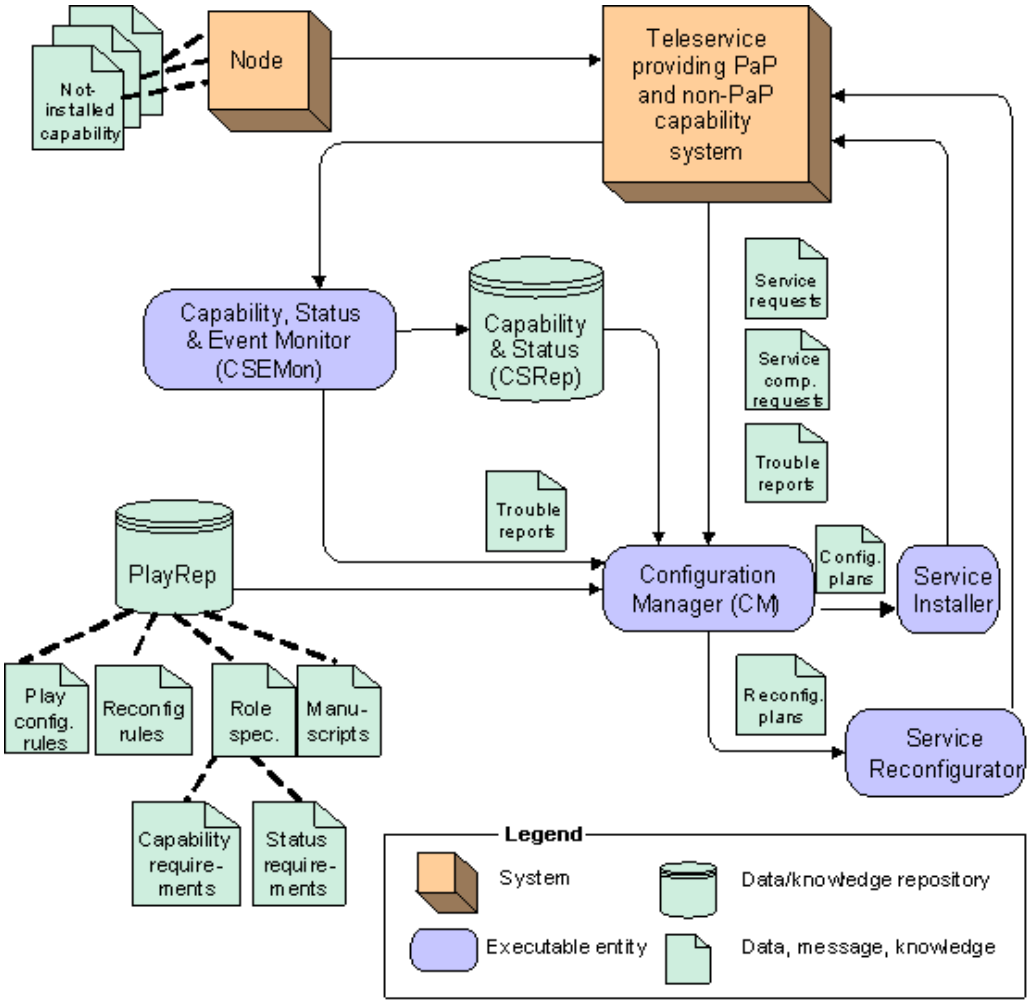


Figure 2-6: Architectural framework for dynamic configuration.

2. *Play Repository (PlayRep)* comprises a set of play definitions defining functional behaviours of particular service systems, each consisting of several actors playing different roles with certain requirements on available capabilities and status. A play definition is an aggregation of the following four specifications:

(i) ***Manuscripts*** define the functional behaviour of each role which not only embraces its internal behaviour, but also the interactions and cooperation with other roles.

(ii) ***Role specifications*** identify, for each role, its requirements on both available capabilities and status.

(iii) ***Play configuration*** rules describe service system configuration constraints, such as specification of the maximum number of roles which are allowed to install at a specific node in order to avoid an overload situation.

(iv) ***Reconfiguration rules*** define application-specific reconfiguration policies for handling substantial reconfiguration-related events.

3. *Capability, Status & Event Monitor (CSEMon)* monitors the system capabilities/status and also maintains the CSRep. Moreover, it listens to certain events indicating changes to the system and its environment, which would prevent the system from getting the desired level of services.
4. *Configuration Manager (CM)* is responsible for: Generation of appropriate configurations for composing new services to be installed in a system, Determination of a location for executing a particular role, and the Computation of reconfiguration schemes for dynamic reconfiguration of existing service systems.
5. *Service Installer* is responsible for the installation of a service into the system by creating corresponding actors for execution of certain roles, according to an obtained play configuration generated by the CM. Allocation of capabilities as well as instantiation of a manuscript for each role are also performed by this entity.
6. *Service Reconfigurator* initiates and performs reconfiguration of a service system based on an obtained reconfiguration plan.

2.3 XML-based Framework for Dynamic Service Management

The framework presented in this section is the TAPAS Dynamic Service Management architecture. This framework has been implemented and it is fully specified in [13]. The Dynamic Service Management framework is presented to show the working scenario of the Selection Engine. The Selection Engine, introduced in section 1.4 and further explained in section 2.4, is the computing mechanism of the previously

presented Dynamic Configuration and the Dynamic Service Management frameworks. The implemented Web services architecture in TAPAS handles the application integration issue, and the Selection Engine is the application to be integrated in the TAPAS platform.

This section has been divided into the same subsections as the previous one. First, subsection 2.3.1 presents the concepts of the Dynamic Service Management framework. Then, in subsection 2.3.2, the framework itself is described.

The main idea in the Dynamic Service Management framework is to use XML behavior specifications, which are basically state machine specifications, with generalized action types as the *Service Specifications*. The system resources are represented by the so-called *Capability* and *Status*, which characterize all the information related to resources, functions and data inherent to a particular node and may be used by a service component to achieve its functionality. *Service Adaptation* is achieved by allowing the service components to modify their functionality, or the code they run, dynamically by requesting changes to their service specification. The framework uses web services to manage the availability of service components, and the communication between them.

2.3.1 Service Management concepts

A basic assumption used in the *Dynamic Service Management* framework, is the notion of *Capability*. In short, *Capability* is a concept representing and abstracting all the information related to functions, resources and data required by the Role-Figures, in order to achieve their desired functionality. Role-Figures are specified as a state machine model that achieves tasks by performing or executing actions. A capability-based Role-Figure specification is a specification that performs such actions, and imposes further requirements on whether an action is still executable if its demand for capabilities is not met at a given time.

Another important concept is *Status*. It comprises observable counting measures, measures for Quality of Service (QoS), or calculated predicates related to these counts and calculated measures. It reflects the resulting state of the system, which cannot directly be changed and negotiated. Basically, status information show, at a certain

point of time, the situation in the system with respect to the actual number of nodes, executing programs, number of users, traffic situation, etc.

One more concept here is *Capability Categories*. It is service designers' work to map actions, according to Action Types, to executable routines provided by service manufacturers. The classification of these executable routines is referred to as *Capability Categories*, where each category represents a capability set indicating operating circumstances and capability requirements that the routines are demanding for or working within.

Role-Figure specifications and Action Library are made available on a Web-server at some well-known address, and can be dynamically downloaded to any node for instantiation. Moreover, a State Machine execution support is needed to execute the XML-based service specification. We will refer to this execution support by State Machine Interpreter (SMI), and assign it the responsibility of managing and executing the Role-Figure specifications and the linking of action definitions with their implementations. The SMI executes the Role-Figure specifications by invoking the action codes with matching action type and capability requirements, i.e., the codes will be selected when the offered Capability Category matches the required Capability Category for the specific action type.

2.3.2 The Framework

The implementation of this framework has been conducted within the TAPAS project and its core platform that provided the basis for the implemented system. The framework for dynamic service management is illustrated in Figure 2-7. The components of the framework are:

1. *Play Repository:* a data base that contains the service definitions and includes: Role-Figure Specifications, Selection Rules and Mapping Rules.
2. *Capability and Status Repository (CSRep):* a database that provides a snapshot of the resources of the system. It maintains information on all capability and status data that may affect the execution of the various Role-Figures at different nodes.

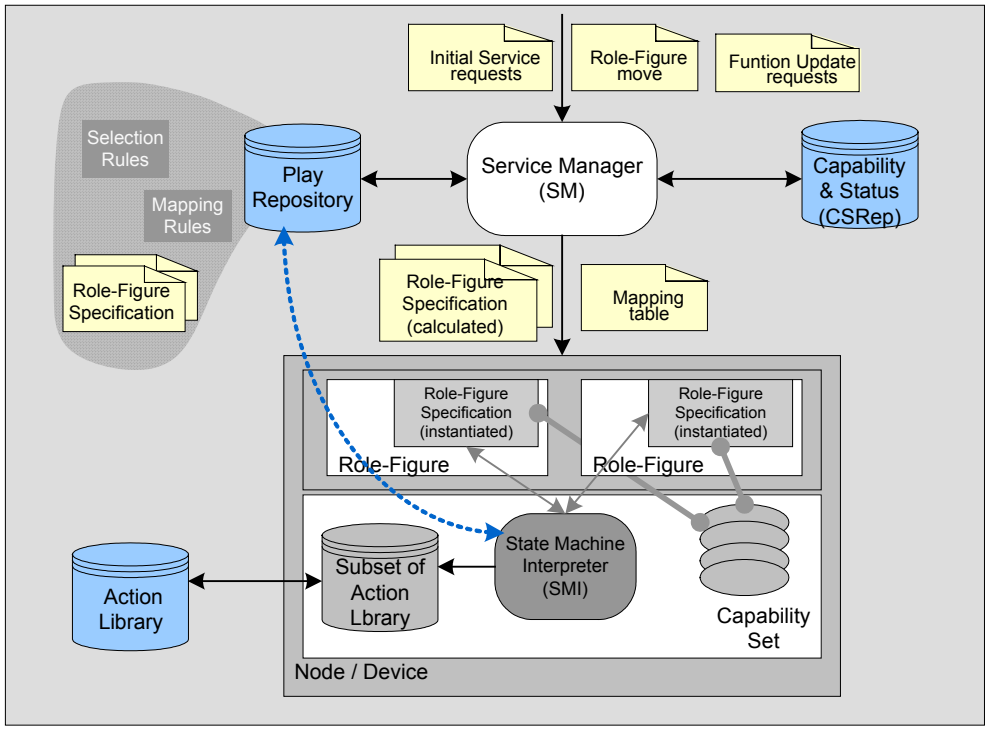


Figure 2-7: Dynamic Service Management Framework.

3. *Action Library:* a database that contains codes for the state machine-based actions. These codes are implemented according to the capability category they require.
4. *Service Manager (SM):* is responsible for the handling of *Initial Service requests* (to instantiate a Role-Figure), *Role-Figure move* (to move an already instantiated Role-Figure from one node to another), and *Function Update requests* (to change the functionality of an already instantiated Role-Figure).
5. *Requests:* supply, on the one hand, the identification of the Role-Figure to be instantiated or modified. On the other hand, they will provide the information that will be taken into consideration during the calculation of the Capability Category. Three types of service requests may be handled by the SM: *Initial Service request*, *Role-Figure move* and *Function Update request*.
6. *Results:* are the outcomes of the calculations performed by SM, which contains a *Calculated Role-Figure Specification* and a *Mapping table*.
7. *State Machine Interpreter (SMI)*: is the primary entity in the framework responsible for the execution of Role-Figures according to *instantiated Role-Figure Specification* sent by the SM.

2.4 The Selection Engine

The reasoning component of both frameworks presented is the Selection Engine. This XET-based engine is responsible for calculating the configuration plans when a service plug-in request is received. The computation mechanism is developed by means of the ET (Equivalent Transformation) paradigm which, in brief, works as follows: Let P and Q be XDD descriptions which model the CSRep and the PlayRep, respectively. Given a message M (either a request message or a trouble report), the CM computes a corresponding (re)configuration plan by simplifying the description P ∪ Q ∪ {M} through repetitive application of semantically-equivalent transformation rules until the description R which contains the computed configuration is obtained. By enforcing this semantic-preservation property of each transformation, the correctness of the computation under this paradigm is always guaranteed [14].

The Selection Engine is founded on the ET paradigm; three declarative-style rule-based programming languages and inference engines: ETC (Equivalent Transformation Compiler), ETI (Equivalent Transformer Interpreter) and XET (XML Equivalent Transformation). Since XET engine can directly operate and reason about XML expressions and XML clauses, it is employed as the computation apparatus to develop an executable engine for reasoning with both dynamic configuration and dynamic service management frameworks.

The interaction with the Selection Engine is done using XML files. But the current TAPAS core platform does not support this kind of communication. The implementation of Web services architecture in TAPAS comes forward to solve this integration problem. With the communication infrastructure presented in this thesis, the TAPAS support platform can interact successfully with the Selection Engine. Therefore, it makes possible the integration of the TAPAS Basic architecture with the TAPAS Dynamic Configuration architecture.

3 Web services architecture in TAPAS

The task performed in this thesis has been a redesign of the TAPAS communication infrastructure to integrate this platform with the Dynamic Configuration architecture. The integration has been achieved by means of the Web services technology. The implementation of Web services architecture in TAPAS provides a feasible approach to the integration issue, i.e. integrating the XET-selection engine into the TAPAS support platform. This XET-selection engine is the main computing component responsible for all the calculations of the (re)configuration plans in the Dynamic Configuration Architecture, and the selection rules and mapping tables in the Dynamic Service Management Architecture.

The work comprises the following subtasks:

1) Implementing Web-services support in the TAPAS communication infrastructure. This task is described in section 3.1.
2) Developing a communication model and an interface for the Selection Engine in order to fit it into the TAPAS support platform. Section 3.2 discusses this task.
3) Providing the extended and needed support to the Director object. This issue is explained in section 3.3
4) Conducting a case-study that comprises experimentation on the TeleSchool application under the TAPAS support platform. This task is introduced in section 3.4 and further developed in chapter 5.

The following sections of this chapter describe the task carried out in each one of the previous points. But the technical details and implementation issues are shown in chapter 4. Tasks 1-4 represent the stages followed to carry out this thesis and state the goals of the thesis. The implementation details behind these tasks are presented separately, in chapter 4, as they are related to the whole thesis and cannot be divided in evaluative phases.

3.1 Web Services support in TAPAS

The first issue in this thesis will be to give Web services support to TAPAS platform in order to be compliant with the currently open Internet standards being used. A TAPAS support system has been already implemented using Java RMI. This technology constitutes the core communication of the implemented support system, allowing the communication and method invocation between service components of a service system. But, because of Internet heterogeneous nature, communication mechanisms must be platform-independent, international, secure, and as lightweight as possible; features that Java RMI does not gather. For this reason, the first change should be done in the communication infrastructure of TAPAS. This change will involve two kinds of communication used in TAPAS. In one hand, there is the communication between service components located at the same node. This communication will be performed using sockets, a more lightweight communication mechanism than the Java RMI used. On the other hand, there is the communication between service components located in different nodes. This looks at three main issues.

1. The communication process runs through different networks crossing firewalls and it is desirable that the service is available from any endpoint located in the Internet.
2. Thus, the service system can be scattered on the Internet, having service components in different networks. A standard mechanism for register and discover the active components is needed.
3. There is also the requirement that a customer, external to the service system, would want to use the service offered. This customer, maybe from a non Java platform, should be able to develop its own client application able to consume the service deployed in TAPAS. Therefore, some information on how to implement such a client is needed. This description information should be available in a standard way so any user can make proper use of it.

The architecture that comes forward as a natural choice and covers all these three issues is the Web services architecture.

The Web services framework is divided into three areas — communication protocols, service descriptions, and service discovery — and specifications are being developed for each. The specifications that are currently the most salient and stable in each area are:

- the Simple Object Access Protocol (SOAP, www.w3.org/2000/xp) which enables communication among Web services;
- the Web Services Description Language (WSDL, www.w3.org/TR/wsdl.html), which provides a formal, computer-readable description of Web services;
- and the Universal Description, Discovery, and Integration (UDDI, www.uddi.org) directory, which is a registry of Web services descriptions.

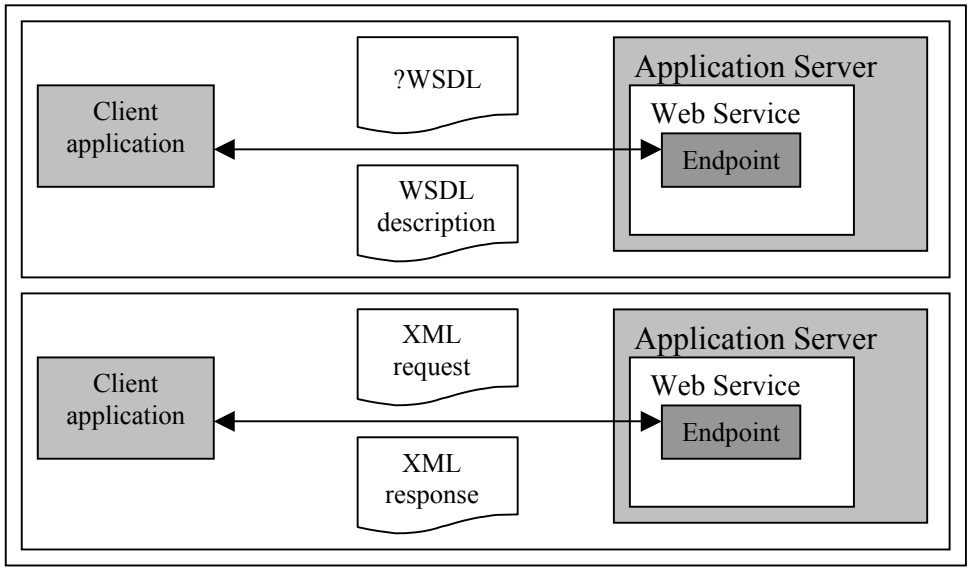


Figure 3-1: SOAP-based Web Service

Figure 3-1 represents a common SOAP-based Web service. The server offers a service description, using WSDL, which will be fetched by the client application. The client application, with the information given in the WSDL service description, will access the service endpoint using XML request. These XML requests can carry an embedded Remote *Procedure Call* (RPC), compliant with the SOAP specification for RPC, which is able to run a method in the endpoint service; and the way to do it is also described in the WSDL description. In this communication process the response to the XML request is also of XML type. Furthermore, as it is a synchronous communication the response will be sent back in the same http connection.

In appendix A, an introduction to Web services is given providing the reader with a complete overview on these specifications and how they are related and used in the Web services technology.

The task done in this section involves extending the functionality of the Plug-and-Play Node Execution Support layer of the TAPAS layered model. This layer represents the basic execution and communication support for running a Plug-and-Play application at a node. In order to use Web services technology, the PNES must be compliant with the web services architecture; this is to be able to communicate with an UDDI registry server, for registering and discovering tasks, and to communicate with other nodes using SOAP messages. Thus, the PNES must implement some registering mechanisms to register the TAPAS Server node at the startup. Other mechanisms implemented are the ones for discovering other active TAPAS Servers scattered on Internet, and finally, it must implement some communication mechanisms to send and handle SOAP messages to and from other nodes running a TAPAS Server. As far as the TAPAS support platform has been developed using Java programming language, all the communication mechanism mentioned before have been implemented using the same programming language, modifying the already existing classes and creating new ones. The most important modifications made to the existing classes are due to the problem of the serialization of Java objects.

The information encoded in the SOAP message must be comprehensible by any client application, even if it is not Java based. For this reason, the Java objects used within the TAPAS support platform cannot be sent across the net without serialization. This serialization intends to convert the properties of the class into XML compliant data format; this is using standardized data types and the appropriated metadata. Due to this serialization work, any system based in any platform can process what it is being sent and it can create a client able to interoperate with the service offered by our TAPAS server.

For the implementation of this task, the Apache Axis [26] and the Java Registry [25] server have been used. Both applications need a server container for which the Jakarta Tomcat server has been chosen. The Apache Axis is a SOAP aware server able to send and receive SOAP messages, and handle them in the correct way. Besides this,

Apache Axis gives support for WSDL description of the deployed service in a very easy way, creating the service WSDL description and the associated Java classes automatically. The Java Registry server is used as a UDDI based registry that allows service components to register themselves and discover other ones active in the network.

3.2 Selection Engine integration in TAPAS

The second issue considered in this thesis is the design of a communication model and the specification of an interface for the Selection Engine. The purpose of this work is to fit this computing mechanism, a XET-based selection engine, into the TAPAS platform.

The *Configuration Manager* form the *Dynamic Configuration* framework, as reported in section 2.2.2, is responsible for the generation of appropriated configurations for composing new services to be installed in a Plug-and-Play system, determination of a location for executing a particular role and computation of reconfiguration schemes for dynamic reconfiguration of existing systems. In the *Dynamic Service Management* framework a similar element is defined, the *Service Manager* (SM). As stated in section 2.3.2, this element is responsible for instantiation of a Role-Figure, moving an already instantiated Role-Figure form one node to another and changing the functionality of an already instantiated Role-Figure. Both elements make use of a computation mechanism developed by means of the *XDD* (*XML Declarative Description*) [16] paradigm, and uses the XET engine [27] that employs the XML syntax and the Equivalent Transformation paradigm. The XET engine can directly operate and reason about XML expressions and XML clauses, for this reason it is employed as the computation apparatus to develop an executable engine for reasoning with the dynamic configuration architecture. This engine represents the *Selection Engine* introduced in section 2.4.

What has been done in this thesis is the creation of a web service interface for the Selection Engine to interact with the DirectorActor Role-Figure of the Plug-and-Play architecture in TAPAS. The interface of the Selection Engine should be able to handle SOAP requests as this is the standard communication protocol for the Web services framework. However, as reported before, the selection engine consumes XML

expressions and clauses so it is required for the client application to send XML requests to it. The SOAP with Attachments API for Java (SAAJ) [25] specification comes forward as the most suitable solution to handle this. As far as the whole support platform for TAPAS is implemented using Java, the SAAJ specification permits to send any kind of information attached to a SOAP message. This way, it is possible to send a XML request attached to the SOAP message that will be received by the *Selection Engine* service interface.

The implementation of the *Selection Engine* interface has been developed using Java Servlet technology and it uses the *XET Engine* provided in [27]. This XET-based engine comprises a Java library containing all compiled classes that allow the execution of the methods needed to compile and calculate the XML clauses and expressions.

The client application, in this implemented model, will send an XML request attached to a SOAP message to the address where the Selection Engine is running, this address is specified at the configuration file. On this address there is a Java Servlet waiting for http connections, and using SAAJ functionality this Servlet will be able to handle the SOAP message that ships with the http request. The SOAP message will be extracted and the attached XML file will be processed by the Selection Engine. This process involves the execution of the received request according to some defined rules and data. These rules and data are XML files accessible by the Selection Engine and it will use them to get the results in the form of an XML file too. The resulting XML file will be attached to a SOAP message and this one will be sent encoded inside the http response.

3.3 Extended support for the Director role-figure

The third issue handled in this report is providing the extended and needed support for the Director role-figure.

The Director role-figure defined in [4] is an important Plug-and-Play object functionality necessary to initialize any play. The director guides actors in the plug-in phase as well as in the plug-out phase.

At the *Dynamic Service Management* framework, there is the *State Machine Interpreter* which is responsible for executing the results given by the *Service Manager*. At the *Dynamic Configuration* framework there is the *Service Installer*, which is responsible for the installation of a service into the Plug-and-Play system according to an obtained play configuration generated by the *Configuration Manager*. Both components are responsible for carrying out the results calculated by the Selection Engine. However, according to the TAPAS support platform, the executor of this kind of tasks is the DirectorActor Role-Figure. The Role-Figure of the Director is the only responsible for the plug-in and instantiation of actors (role-figures).

The solution proposed in this thesis comprises giving the Director Role-Figure an extended support for interpreting the results given by the Selection Engine. This is similar to the task performed by the SMI and the SI; somehow the functionality of these components is integrated into the Director role-figure. Therefore the Director will be the responsible for interpreting the received results and execute them within the TAPAS support platform. The calculated configuration results are encoded using XML and are received by the Director attached to a SOAP message that ships with an http response as a result of an http request to the Selection Engine. Making use of the extended support, the Director must be able to read this XML file, select the most suitable configuration solution and execute it. This execution comprises the creation of corresponding actors for execution of certain roles - if we are working within the Dynamic Configuration framework, or carrying out the instantiated *Role-Figure Specification* sent by the SM - if we are within the *Dynamic Service Management* framework.

The implementation of this issue has been developed using J2SE 1.4.2 SDK, creating new Java classes to extend the support for the Director Role-Figure. The reading of the XML files has been implemented using the Simple API for XML (SAX) [25] provided by the Java API for XML Processing (JAXP) [25]. A SAX parser reads the file and it executes the calculated service configuration according to the conditions and pattern specified in a java class that depends on the service requested to the Selection Engine.

3.4 Conducting a case-study on existing TAPAS application

This task is about conducting a case-study that comprises experimentation with different scenarios on some existing TAPAS application, TeleSchool application has been chosen for this purpose.

TeleSchool is an example application built to demonstrate the use of the TAPAS platform. An implementation of the application already exists that builds on the Basic TAPAS Architecture. As the name of the application indicates, the functionality of the application is related to schools and network based learning.

In TeleSchool students and teachers attaches to a school to get access to services. The services provided are utilized to perform real time lectures, review stored lectures and to allow communication between students and teachers. The services may include distribution of multimedia communication.

Four different roles have been defined for the play real time lecture. These are shown in Table 1. In the current TeleSchool implementation each of these roles are represented by a Java class containing a description of the role's manuscript, and information related to role specifications is specified when requesting actor plug-ins.

Table 1: TeleSchool roles

Role	Description
ShoolRTLServer	Provides functionality specific for real time lectures.
SchoolServer	Defines the behaviour of the server for all clients running TeleSchool.
SchoolClient	Defines the behaviour of students and teachers.
SchoolUserInterface	Presents the user interface for the students and teachers.

The purpose of this example scenario is to demonstrate the validity of the proposed communication model. The experimentation with this communication infrastructure comprises the following issues:

- Registering the TAPAS server node in a UDDI Registry server at the startup.
- Communication with the Selection Engine in order to configure the plug-in of the TeleSchool service. This task involves the communication process between the Director Role-Figure and the Selection Engine which will test

the communication using the SOAP API for Attachments in Java; sending a XML query attached to a SOAP request and getting back the XML result.

- Testing the extended support of the Director Role-Figure. Once the result has been received, the Director must read it and execute the appropriate actions in order to fulfill the configuration plans calculated by the selection engine. The configuration plans may include the instantiation of a new Role-Figure at a certain node, but first, the Director should make sure that the destination node is an active TAPAS server. For this it is necessary a discover mechanism.
- Discovering mechanism to find active TAPAS server nodes at the network. Before sending the plug-in request to a node the Director must be sure that this node is running a valid TAPAS support system. The implemented mechanism allows querying the Registry Server for registered nodes at the network and getting the location of a WSDL description for the offered web service.
- Communication between TAPAS Server nodes, this is between PNES entities, using the existing Web open standards. In this framework this is done exchanging SOAP messages in a web services architecture, encoding the parameters and all necessary information using standard data types and the Remote Procedure Call specification given with SOAP.

4 Implementation issues

The implementation of Web services architecture in TAPAS has been conducted within the TAPAS project and its core platform that provides the basis for the implemented system. The implementation has been developed using JAVA and Web technologies as a means for service definition, update and discovery. The main difference with the original TAPAS support platform is the application of an all-web-services node registry and communication model, which achieves a XML-based architecture with application integration support.

Java Web Services Developer Pack (Java WSDP) has been used to develop the main communication parts of the framework, while Apache Axis has been used as a SOAP server. In this regard, nodes running the platform will have an entity that supports Web Services requests and replies. This modified communication layer needs a Web Server, in this case is the Tomcat server provided with the Java WSDP, and a SOAP server which can handle the SOAP requests generated in the communication between nodes and sends back the reply in a synchronous mode. The Registry Server provided with the Java WSDP is used to register nodes executing Role-Figures. SOAP messages with attachments are used to send and receive SOAP messages with an attached query in XML format.

Next section will focus in giving a distributed communication model for the TAPAS platform with support for a *Dynamic Service Management*. Besides, the distributed solution presented here will be compared to the current distributed solution used in the TAPAS Basic architecture.

The Plug-and-Play distributed solution for the TAPAS platform used Java RMI to run methods on Java Objects (JO) located in different Java Virtual Machines (JVM). The proposed TAPAS communication model avoids RMI. The communication between JO in different JVM in the same node is done using sockets and the communication between different nodes uses the SOAP protocol. The main problem to face is that the TAPAS architecture claims for a peer to peer communication, and this is a problem when using Web services. Web services are conceived as client-server communication model, but in the TAPAS model every node acts as client and server. When an actor

at any node requests a service from another node the local PNES will act as a Web service client. This issue can be described in short as follows: first, the PNES instance at a node will request the registry to know if the receiver node is an active TAPAS server. If success, it will query the remote node using the RPC specification encoded in a SOAP request. Notice that this node will be running server able to handle this SOAP request. At the same time, this node acting as client can receive a SOAP request from a remote node trying to access the active service, and then it will be acting as a server application. So both client and server side code should be present at every node acting as a TAPAS server. This has the inconvenient that a SOAP aware server must be running at every node and the TAPAS service should be correctly deployed in it.

This chapter is divided into four sections. The first section 4.1 describes the proposed communication infrastructure for the TAPAS Platform to bear with Web services architecture. Section 4.2 explains the technical details of adding Web services support to the TAPAS core platform and how to implement the Web services architecture in TAPAS. The interaction process between the Director and the Selection Engine is presented in section 4.3. Finally, there are some problems regarding the interoperability issue when using SOAP for application integration. This is described in section 4.4.

4.1 The proposed communication infrastructure

The main changes made to the distributed model are the communication protocol and the way in which the synchronous communication among the JO instances is done. The original communication model is based on Java RMI; this is used to run methods in remote JO located in different JVMs. There are two situations in which this happens; the communication between actors belonging to different PAS at the same node – notice that each PAS runs on a different JVM, and the communication between actors located in different nodes. When running at the same node, instead of RMI it is possible to use sockets. And for the communication between nodes, this is, between PNES, the solution proposed uses SOAP and UDDI. As stated in the Web Services introduction, SOAP supports a standard for RPC – this is necessary to replace the Java RMI function, and with UDDI we can publish, discover and query the active PNES running in the network – this is necessary to replace the RMI registry provided by

Java RMI. With this new distributed solution it is possible to avoid using RMI and integrate the Web services technology to the TAPAS architecture.

The presentation of the proposed communication infrastructure is divided into two main sections; first it is presented the Synchronous Communication Model in section 4.1.1 including the implementation details. Next section, the Entity Registry 4.1.2, illustrates how a registry server has been integrated into the TAPAS support platform allowing the registration and discovery of active TAPAS Servers at the network.

4.1.1 The Synchronous communication model

The proposed synchronous communication model for the TAPAS platform with Web services architecture is shown in Figure 4-1. This model avoids using Java RMI so the communication can be established with any non-Java platform using open Web standards.

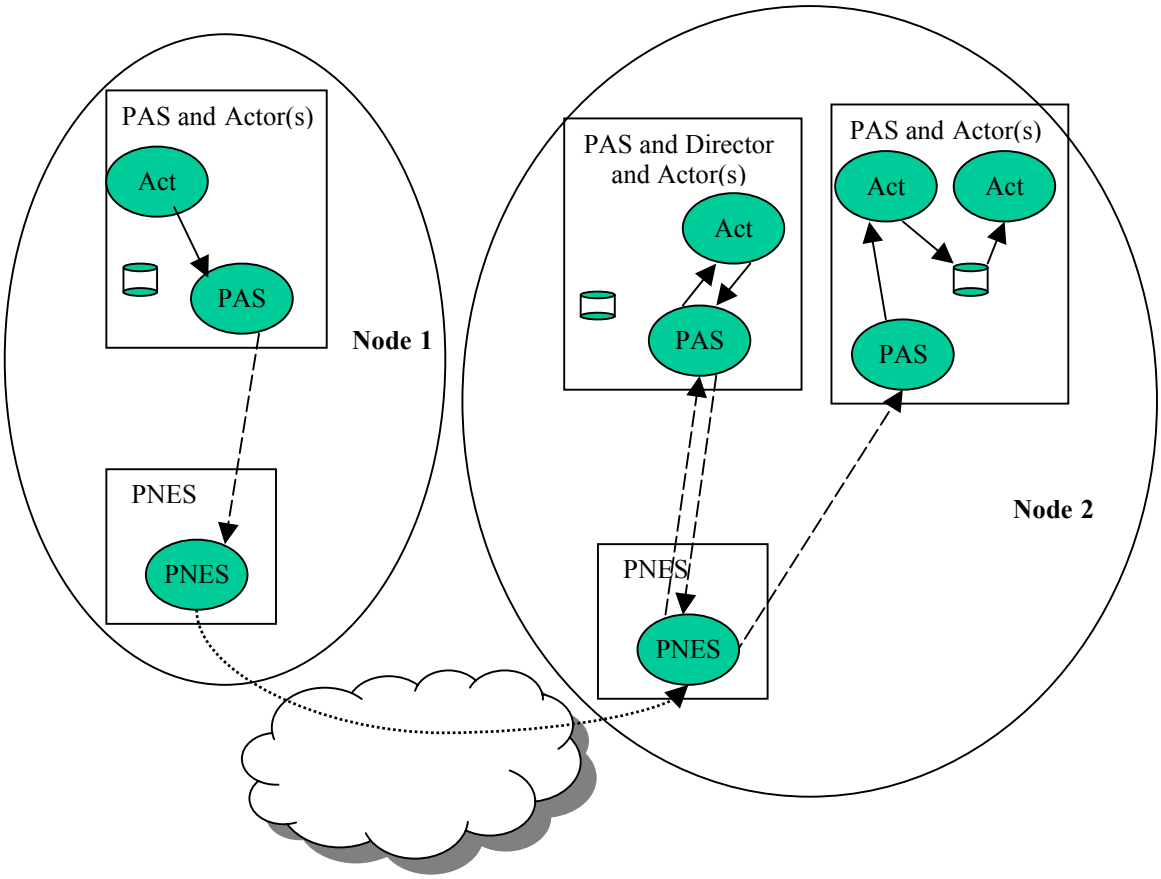
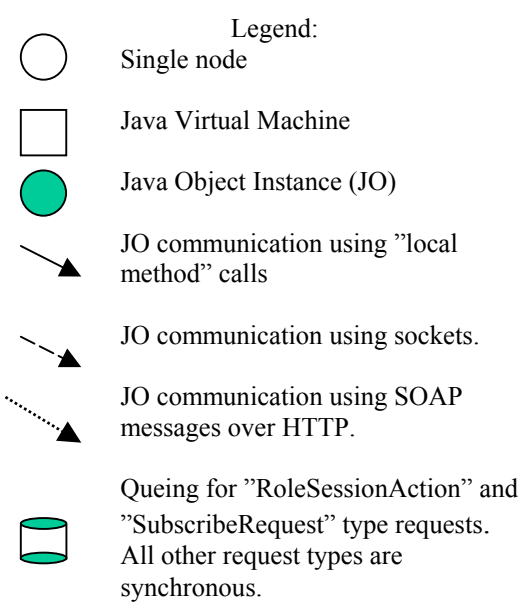


Figure 4-1: The Synchronous Communication Model.

Communication among actors belonging to the same PAS instance, the same JVM, is still done using local method calls. PAS communicates directly only with PNES instance at the same node. This communication between two JVM is done now using a socket connection. And the communication between two nodes, two PNES instances, uses the SOAP standard for RPC.



As shown in Figure 4-1, the communication between the actors and its PAS within the same JVM is done using local method calls. This is in the same way as it was done in the original Plug-and-Play distributed solution. Regarding the PAS, this communicates directly with the one and only PNES at the same node, by using socket connections. All communication between PAS is carried out through the PNES. In the Plug-and-Play distributed solution remote method calls were used for the communication between these instances located in different JVM, and there was a direct communication between PAS located at the same node. PNES communicates with other PNES at other nodes by using the SOAP specification for Remote Procedure Calls (RPC). This communication model is illustrated in Figure 4-1.

Regarding the use of sockets between JVMs located at the same node instead of Remote Method Invocation, is a matter of lightening the communication process. Distributed object-based applications can be easily developed using Java Remote Method Invocation (RMI). The simplicity of RMI, however, comes at the expense of network communication overhead. Low-level sockets can be used to develop client/server systems, but since most Java I/O classes are not object friendly, we need the object serialization as the mechanism that allows read/write full-blown objects to byte streams. Combining low-level sockets and object serialization gives a powerful, efficient alternative to RMI that enables to transport objects over sockets and

overcome the overhead incurred in using RMI. A detailed implementation using sockets for this communication is given in Figure 4-2.

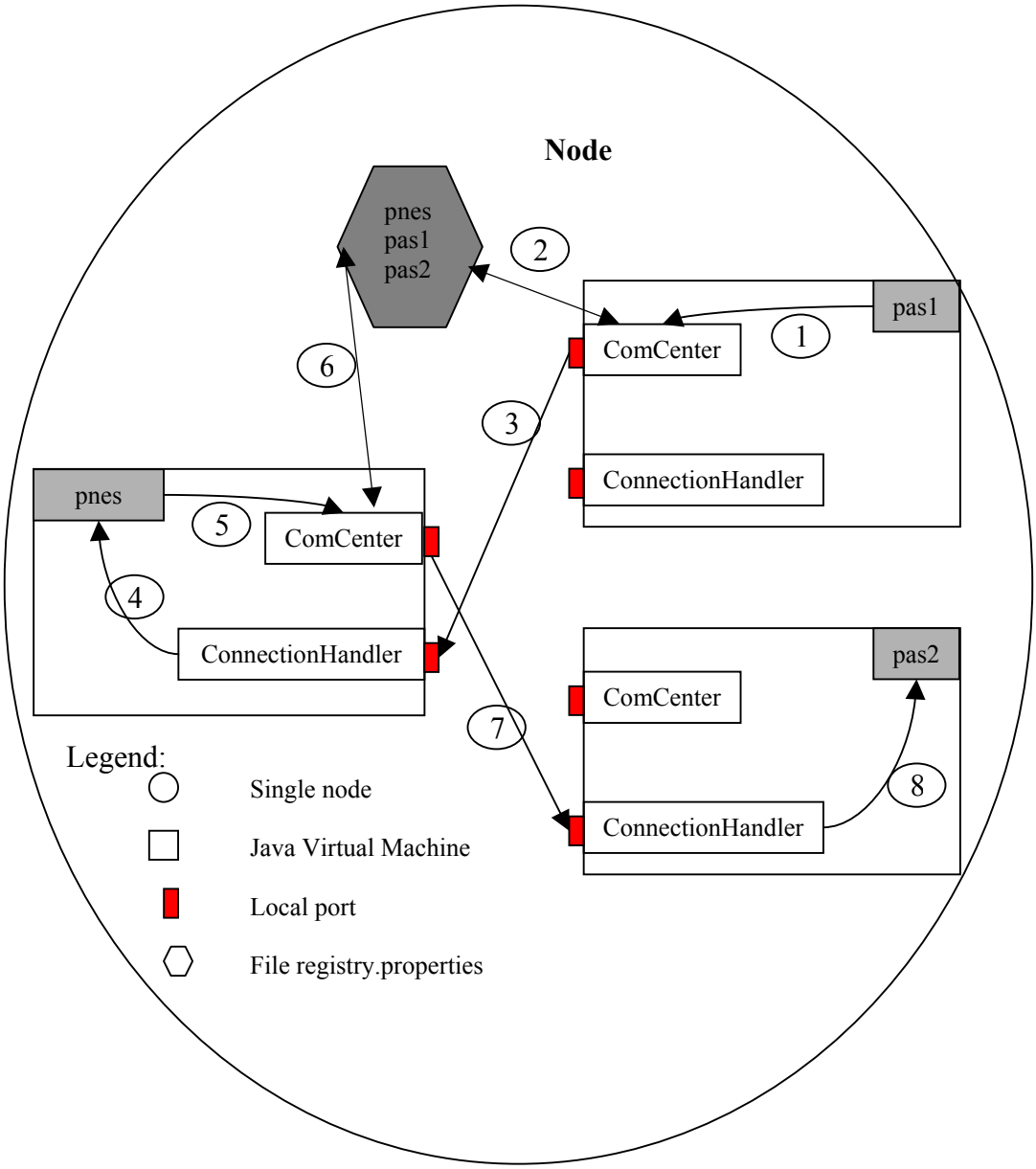


Figure 4-2: Socket communication model

PAS communicates directly only with the one and only PNES at the same node using socket connections. There is also a registry.properties file that keeps a mapping table with all the active PAS and PNES instances and their corresponding local port where the server socket is listening.

When using RMI it is possible to invoke directly a method in a remote object as if it was at the same JVM, but when using sockets only byte streams can be send through them. So this information has not any kind of format and does not belong to any Java

data type. The developer of the application is responsible for giving any format to this raw information sent across the network. For instance, to send Java objects using sockets it is necessary to serialize them at the client side and deserialize at the server side - at the receiver.

A communication model and its protocol have been designed and implemented using sockets in order to emulate the same functionality given with RMI; this is executing methods on remote objects and keeping a registry where the application can look for the different PAS and PNES instances running at the same node. Figure 4-2 illustrates the communication protocol in steps 1-8. These steps are explained bellow:

1. A PAS instance initiates the communication process when it receives a request from one of its actors and the receiver actor does not belong to the same PAS instance. This request contains the required *request parameters* needed in any standard Plug-and-Play request type, this is a Java object called RequestPars.
2. The entity in charge of getting the port number of the PNES instance, opening the client socket and serializing the RequestPars object into a byte stream is the ComCenter. Before establishing the connection, the ComCenter should check if there is any active PNES instance. In the original distributed Plug-and-Play solution this was done using the rmiregistry, all instances were registered in the registry at the start up. And any client entity could retrieve the receiver's interface and run any method on the remote object. Now, there is a mapping table with all the active instances at the node and their corresponding port number where a server socket is listening from. In that way, if the receiver entity, PAS or PNES, has not been registered it will not appear in the mapping table and it will not be possible to connect with it. This mapping table is implemented as a properties file called registry.properties. This file is created every time that the PNES instance is started, and deleted when exiting.
3. A PAS instance communicates directly only with the one and only PNES instance at the same node. The PNES instance is listening from a default local port specified in the configuration properties file and is waiting for remote incoming connections. All PAS requests will be addressed to this port number once the ComCenter has checked that there is an active PNES instance registered at the mapping table. The ComCenter will get the port number

associated and will send the request parameters of the PAS request through a socket connection. The socket connection remains open until the corresponding ResultRequest object is received. This ResultRequest object is the standard response for all Plug-and-Play requests and it is received as a byte stream, so the ComCenter has to deserialize it and send it to the initiator of the request.

4. The ConnectionHandler is the entity responsible for handling the incoming socket connections. There is a server socket listening from the specified port in the mapping table. All the incoming connections come from PAS instances, any other PNES can connect directly with this one using a socket connection. The ConnectionHandler is expecting a RequestPars object, so it will deserialize the received stream of bytes from the socket into a RequestPars object. And then it will invoke the appropriated method on the PNES instance using a local method call with the given parameters for the request.
5. If the specified receiver at the request parameters is an actor at the same node, the communication with it will be also using sockets. But if the receiver actor belongs to a remote PAS, not located at the same node, the communication will be using the SOAP model. In this case, the actor belongs to a second PAS instance, pas2, running at the same node.
6. The ComCenter will look for the port number associated to the pas2 instance at the mapping table. This port number will be used to establish a new socket connection with the receiver PAS.
7. As in step number 3, the ComCenter will serialize a RequestPars object into a byte stream and send it using a socket connection. This connection will remain open until a ResultRequest object is deserialized out from the socket stream.
8. The same as in number 4, the ConnectionHandler will get the RequestPars object and invoke the appropriate method on the PAS instance using a local method call. The response of this call is a ResultRequest object that will be returned to the client socket side. As it is shown, all the connections are synchronous. So this ResultRequest will go all the way back to the originator of the first request, in this case the pas1 instance.

In step 5 it was said that the receiver actor could be located at a different node than the one of the current PNES. In this case a SOAP communication model is used for

the interaction of two different PNES belonging to different nodes. The original distributed Plug-and-Play solution used RMI to manage this communication between nodes, just using remote method calls. The PNES queried the rmiregistry of the remote node and got an interface of the remote PNES instance. This interface was used then to invoke methods as if it was a local method call.

In order to avoid using RMI, the proposed distributed Plug-and-Play solution uses the SOAP specification for RPC and responses. This allows remote procedure calls to use SOAP as a wrapping protocol and sending them using HTTP as a transport protocol. The communication model using SOAP presents two main issues; using custom classes in SOAP – described in section 4.1.1.1, and how to connect the SOAP aware server with the running PNES instance at the node, section 4.1.1.2.

4.1.1.1 Serializing the RequestPars Java object.

The first issue is closely related to the interoperability matter. The Java objects used in the TAPAS model should be serialized and deserialized before creating the SOAP request. It is not enough just serializing the Java object into a byte stream and sending it as raw information. In this case, only Java compatible platform systems would be able to deserialize the information and get the desired object out of the data stream. The information sent in the SOAP request must be XML compliant, so any system using any platform can understand what it is being sent and it can create a client application able to interoperate with the service offered by the Plug-and-Play server.

Since SOAP is a data transport, we are only interested in the properties of the class. This means that all attributes of the objects being sent with SOAP should be encoded using XML standard data types which are supported by all SOAP implementations.

One common way to express the properties of a Java class is to use the JavaBeans design patterns. These patterns specify a naming convention to be used for the class's access methods. The methods used for getting and setting property values should conform to the standard design pattern for properties. These methods are allowed to throw checked exceptions if desired, but this is optional. The method signatures are as follows:

```
public void set<PropertyName>(<PropertyType> value);
public <PropertyType> get<PropertyName>(  );
```

The existence of a matching pair of methods that conform to this pattern represents a read/write property with the name <PropertyName> of the type <PropertyType>. If only the get method exists, the property is considered to be read only, and if only the set method exists the property is considered to be write only. In the case where the <PropertyType> is boolean, the get method can be replaced or augmented with a method that uses the following signature:

```
public boolean is<PropertyName>(  );
```

If this pattern is followed for naming property accessors, the accessor methods can be determined at runtime by using the Java reflection mechanism. This is a convenient way for SOAP implementations to access the data values of a Java class instance in order to serialize the data in a SOAP message. Following a well-established naming convention will avoid any problem using custom classes in SOAP.

The JavaBeans design patterns have been applied to all classes required to be serialized. Set and Get methods have been created in every Java class related to the RequestPars class, as this is the parameters object needed in the standard TAPAS request. This is the object that needs to be serialized when sending the SOAP message.

4.1.1.2 Interaction of the SOAP server with the TAPAS PNES

There is a problem with the communication between the SOAP server and the TAPAS PNES running at the node. These entities are located in different machine processes, so it is not possible for the server to invoke directly one method of the PNES using local method calls.

As far as the method to invoke resides in a different JVM it is not possible to use local method calls. The SOAP Server receives the parameters of the request, the RequestPars Java object, but it cannot call the appropriate method of the PNES instance for running the request received. The solution to this problem comes by using socket connections to connect the SOAP Server with the PNES. The SOAP Server

will send the parameters of the request to the PNES using a socket connection. Afterwards the PNES will execute the required actions and the result will be sent back to the SOAP Server using the same open socket connection. The SOAP Server, then, will forward this result to the client of the service. This communication model is illustrated in Figure 4-3 using an example with two PNES running in different Plug-and-Play Server nodes and with its associated SOAP Server.

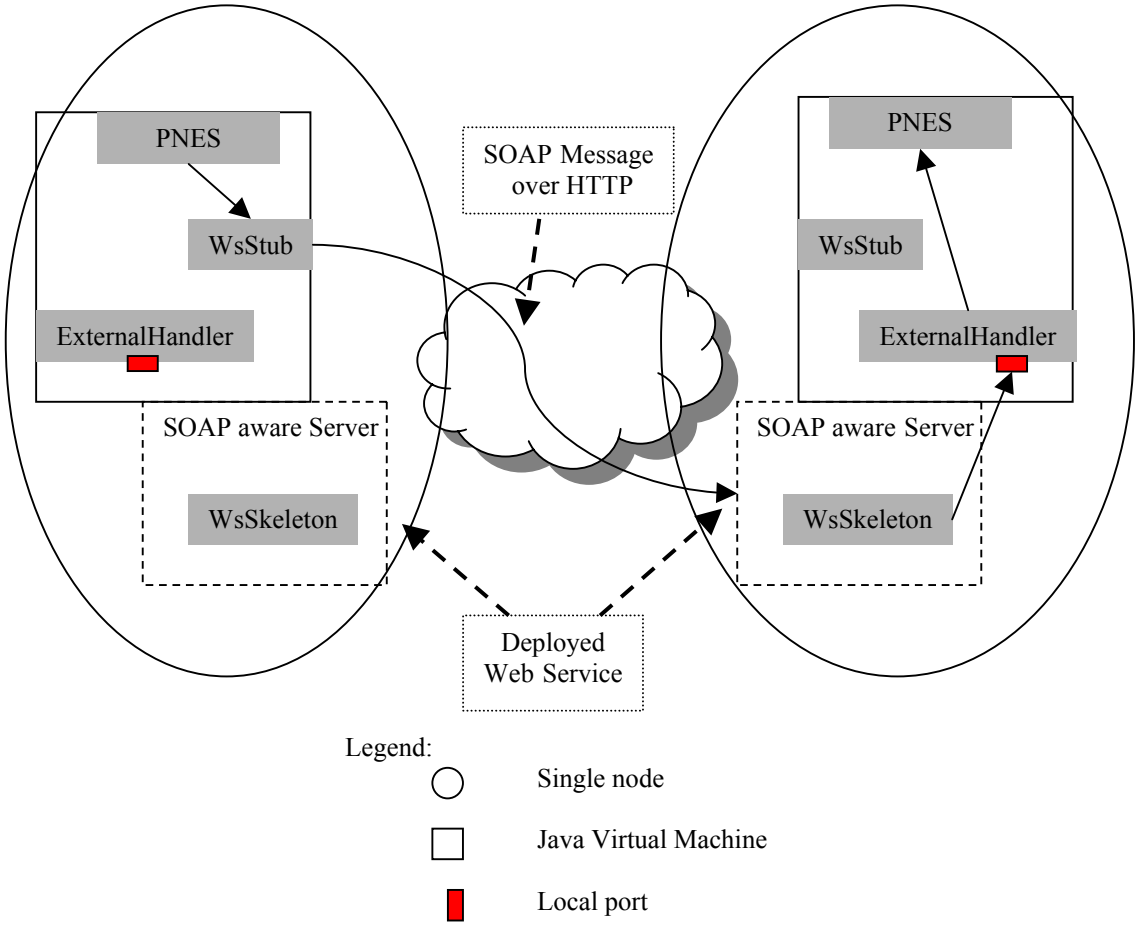


Figure 4-3: SOAP communication model

The PNES uses the WsStub class generated by the SOAP server. This WsStub acts as a local interface for the remote service; this is in the same way as the stub created when using RMI. The local PNES can invoke a method of the remote service directly in this stub as if the service was located at the same JVM. This WsStub is also the client side of the service and it will request the server hosting the Web service, in this case the Plug-and-Play server. The communication process is hidden to the client; the WsStub is responsible of all the settings needed to establish the connection with the application server endpoint. At the server side, the SOAP aware server will handle the

SOAP request and execute the specified remote procedure call with the corresponding parameters. This includes establishing a socket connection with the PNES running at the node and getting the corresponding response. This response will be sent back to the client, the client request was synchronous, so it is waiting for this response. And at the client side, the socket connection is also open waiting for the synchronous response.

4.1.2 The entity registry

The registry mechanism has been modified compared to the mechanism used in the original TAPAS platform. The registry mechanism shown in Figure 4-4 is based on the UDDI service. Web services technology, based on XML messages, is used for discovery and registration procedures.

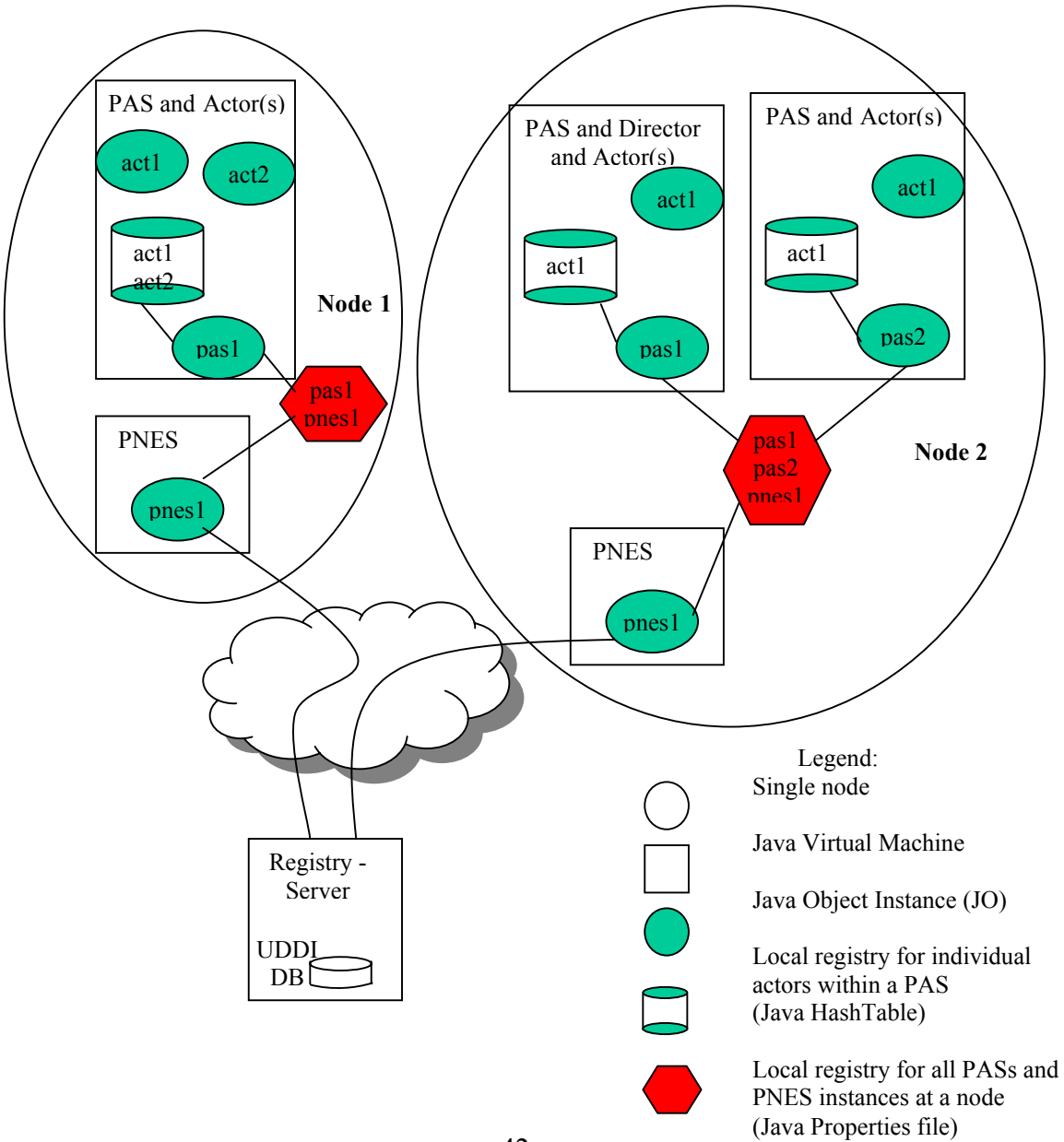


Figure 4-4: The Registry Server with UDDI service

In the current distributed Plug-and-Play solution the *rmiregistry* is used to register and retrieve the remote interfaces of the PNES running at the network. In order to supply this functionality at the presented communication infrastructure in this thesis, it is needed two different registries. One of them is an internal registry implemented as a properties file, this registry exists in every node running as a TAPAS Server and it keeps a local database with all the PAS and PNES running at the node. This registry is the one that was introduced in section 4.1.1, The Synchronous Communication model, when talking about the sockets communication model. As reported then, this local registry is used by the PAS to communicate with other PAS located at different JVMs. The other registry shown in Figure 4-4 is the Registry Server. For the implementation of this communication infrastructure it has been used the Registry Server provided with the Java Web Service Developer Pack, which is modeled as a UDDI-based database at the figure.

This registry is unique for the whole Plug-and-Play service; there is only one registry at the network that keeps a database with all the active Plug-and-Play Server nodes in Internet. The Plug-and-Play Service within the TAPAS project is registered with a defined schema into the registry. This TAPASSchema has a Unique Universal IDentifier (uuid), a key identifier that must be known by every Plug-and-Play Server node in order to register its service with the correct schema and ensure that the registered node is unique in Internet. When a schema is registered for the first time into the registry, it will be provided with a uuid that must be used in the XML definition of the schema. This TAPASSchema XML definition, together with its DTD and the uuid provided, is used at the discovery process of other active nodes, using this unique schema we ensure that the located nodes belong to our own service definition.

The communication with the Registry Server has been developed using the Java API for XML Registries (JAXR) [25] which provides a convenient way to access standard business registries over the Internet. JAXR gives a uniform way to use business registries that are based on open standards (such as ebXML) or industry consortium-led specifications (such as UDDI).

4.2 Adding Web Services to TAPAS

When a different developer wants to create a client application for the TAPAS service it should refer to the WSDL service definition provided by the Web service provider. This description defines all the data types used in the request, the response and the way to call a remote procedure. So the developer can know which data types have been used to encode the Java class attributes that are being sent in the SOAP message. The content of this file is shown in Appendix B.

The Apache Axis library contains two programs for use at build time: WSDL2Java and Java2WSDL. These programs create Java classes from a WSDL description and viceversa.

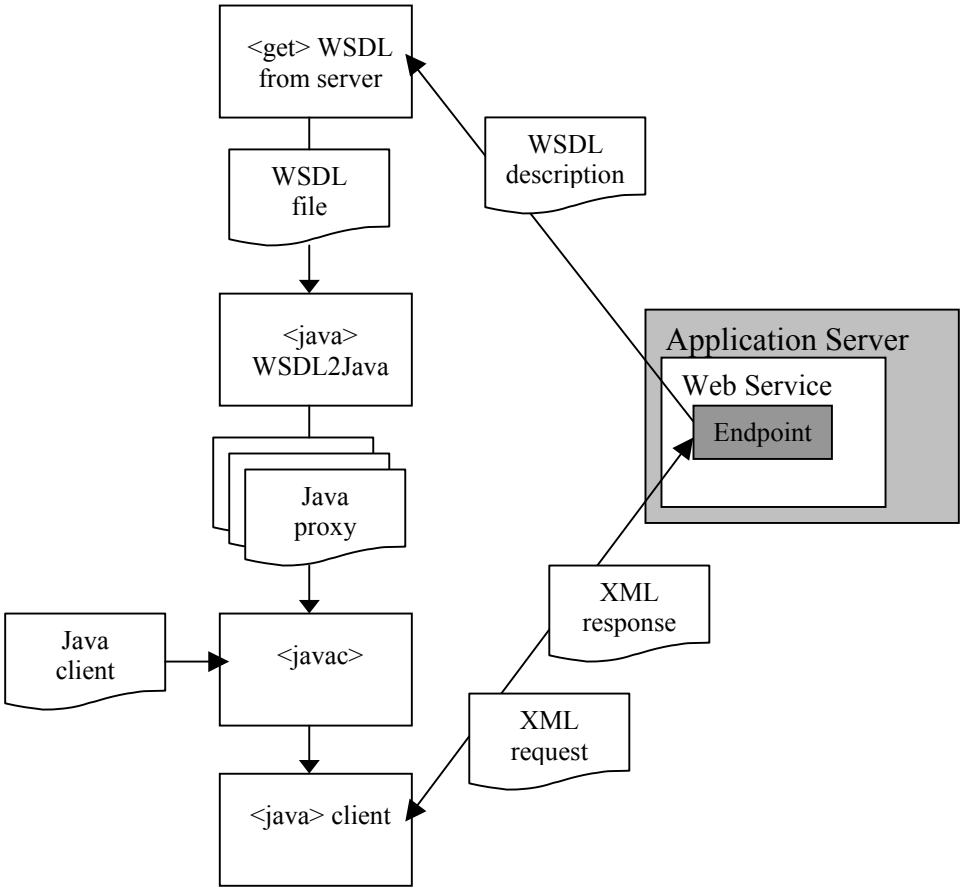


Figure 4-5: Overall workflow of creating a SOAP client application

Figure 4-5 shows how to create a SOAP client application using the WSDL2Java tool provided with the Apache Axis 1.1. As illustrated in the figure, first it is retrieved the WSDL description of the service that we want to create the client for. This WSDL

description is created automatically by the TAPAS Server, and it can be retrieved form the endpoint using a simple http get request. Then, using the WSDL2Java tool a bunch of proxy classes is created. This proxy classes have the same function as the ClientStub created with the "rmic" tool of the Java RMI. They represent a local implementation of the remote service, allowing the client application to make calls on them as if they were the same remote service.

It is developer task to create the appropriated Java client program that will use these generated proxy classes. The developed client application makes use of the service interface provided by the proxy class which is in charge of the whole communication process with the TAPAS Server. All this process is hidden to the developer and to the client application. The communication is realized between these proxy classes and the Web Service endpoint exchanging SOAP messages that encode the XML requests and responses.

4.3 Interaction between Selection Engine and Director

In this section it is given the implementation details of the communication infrastructure used between the Selection Engine and the Director Role-Figure. This communication mechanism was introduced in section 3.2, Design of a communication model and an interface for the Selection Engine so as to fit into the TAPAS support platform.

The communication mechanism is based on the SOAP with Attachments API for Java which provides a standard way to send XML documents over the Internet from the Java platform. It is used mainly for the SOAP messaging that goes on behind the scenes in JAX-RPC and JAXR implementations. And it is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages.

A SAAJ client is a standalone client. That is, it sends point-to-point messages directly to a Web service that is implemented for request-response messaging. Request-response messaging is synchronous, meaning that a request is sent and its response is received in the same operation. A request-response message is sent over a SOAPConnection object via the method SOAPConnection.call, which sends the

message and blocks until it receives a response. A standalone client can operate only in a client role, that is, it can only send requests and receive their responses.

This SAAJ client is integrated into the Director giving it the necessary communication support for interacting with the Selection Engine. The class that implements this SAAJ client at the client side is the ServiceReqClient, and at the server side there is the ServiceReqServlet class. This servlet comprises the web service interface of the Selection Engine, it will receive an XML query attached to the SOAP message and it will run the appropriated methods on the XET-based engine to process the query and get the XML result. The core XET engine of the Selection Engine is provided in a Java library, so its use by the service servlet is straight forward; just running the methods implemented in the package using the given parameters.

The query needed for the Selection Engine and calculated result are XML files. These files are transmitted as attached parts of the SOAP message. Figure 4-6 shows the high-level structure of a SOAP message that has two attachments. And a short description of every component part of the message is also given later to better understand how these attachments ship with the SOAP message.

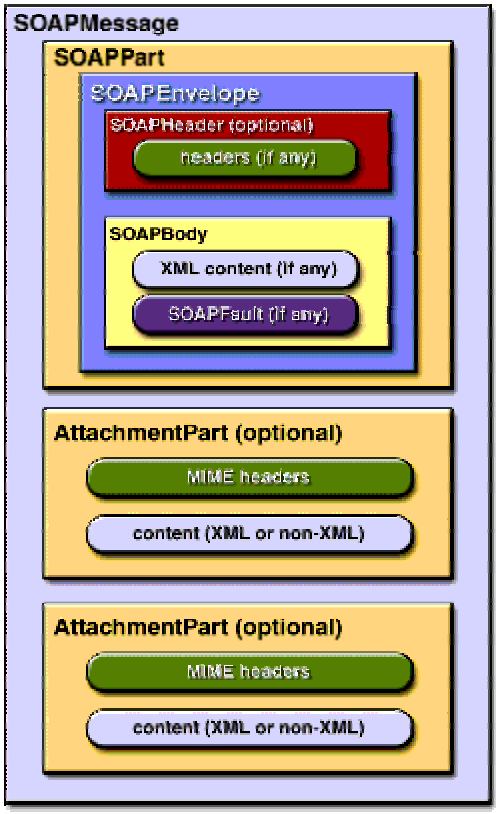


Figure 4-6:SOAPMessage Object with Two AttachmentPart Objects

A SOAPMessage object represents an XML document that is a SOAP message. A SOAPMessage object always has a required SOAP part, and it may also have one or more attachment parts. The SOAP part must always have a SOAPEnvelope object, which must in turn always contain a SOAPBody object. The SOAPEnvelope object may also contain a SOAPHeader object, to which one or more headers can be added. The SOAPBody object can hold XML fragments as the content of the message being sent. If it is wanted to send content that is not in XML format or that is an entire XML document, the message will need to contain an attachment part in addition to the SOAP part. A SOAP message may include one or more attachment parts in addition to the SOAP part. There is no limitation on the content in the attachment part, so it can include images or any other kind of content, including XML fragments and documents. Common types of attachment include sound, picture, and movie data: .mp3, .jpg, and .mpg files.

4.4 Problems related to SOAP

Interoperability is an ongoing issue with SOAP. The developers of SOAP toolkits work on interoperability tests to verify that foundational data types such as strings, integers, Booleans, arrays, and base64 encoded binary data can all be exchanged between clients and servers. But complex types are not yet standardized. Consider the `HashTable` class: Java implements `java.util.HashTable` and .NET has its own implementation in `System.Collections.HashTable`. It is possible to return one of these from a service implemented in one of the languages specified before:

```
public HashTable getEmptyHashTable() {
return new HashTable();
}
```

A client written to use the same toolkit as the service will be able to invoke this SOAP method and get a hashtable back. A client written in another toolkit, or in a different language, will not be able to handle this. If we were writing our server API by coding a WSDL file first and then by writing entry points that implemented this WSDL, we would probably notice that there is no easy way to describe a hashtable; consequently, we would define a clean name-value pair schema to represent it. Because we are developing web services the easy way, by writing the methods and letting the run time do the WSDL generation, we do suffer from the hashtable problem.

5 Experimentation Scenario

In this part it is presented a case-study that comprises experimentation with an existing TAPAS application, the TeleSchool application [15]. It will demonstrate the feasibility of integrating the Selection Engine into the TAPAS support platform under the Dynamic Service Management framework introduced in section 2.3. Web services technology is used for service definition, update and discovery. The Dynamic Service Management is used as the environment where the TeleSchool service will be plugged using the TAPAS support platform with the implemented Web services architecture.

This part is divided into three main sections. First, the experimentation scenario is shown and the goals of the example will be stated. Secondly, some information on how to set up the programs and tools used for the demonstration is given. In the third section, the demonstration itself is presented. It includes some code examples, messages exchanged in the communication process and some configuration properties.

5.1 Describing the scenario

The experimentation scenario, shown in Figure 5-1, is composed of four nodes. Two of the nodes are running the TAPAS support software; this means that there will be an active PNES instance running in each of them, these nodes are called TAPAS Server nodes. The Server node stores the source code of the TAPAS core platform that will be downloaded on the runtime when required by a TAPAS Server node. This Server also performs the functionality of the Play Repository which will store the source code of the TeleSchool play which is the service we want to plug into the system. The remaining node is the Selection Engine node; this node is running a servlet application that performs the functionality of the XET-based Selection Engine. This node is also running a Registry Server for service publishing and discovery.

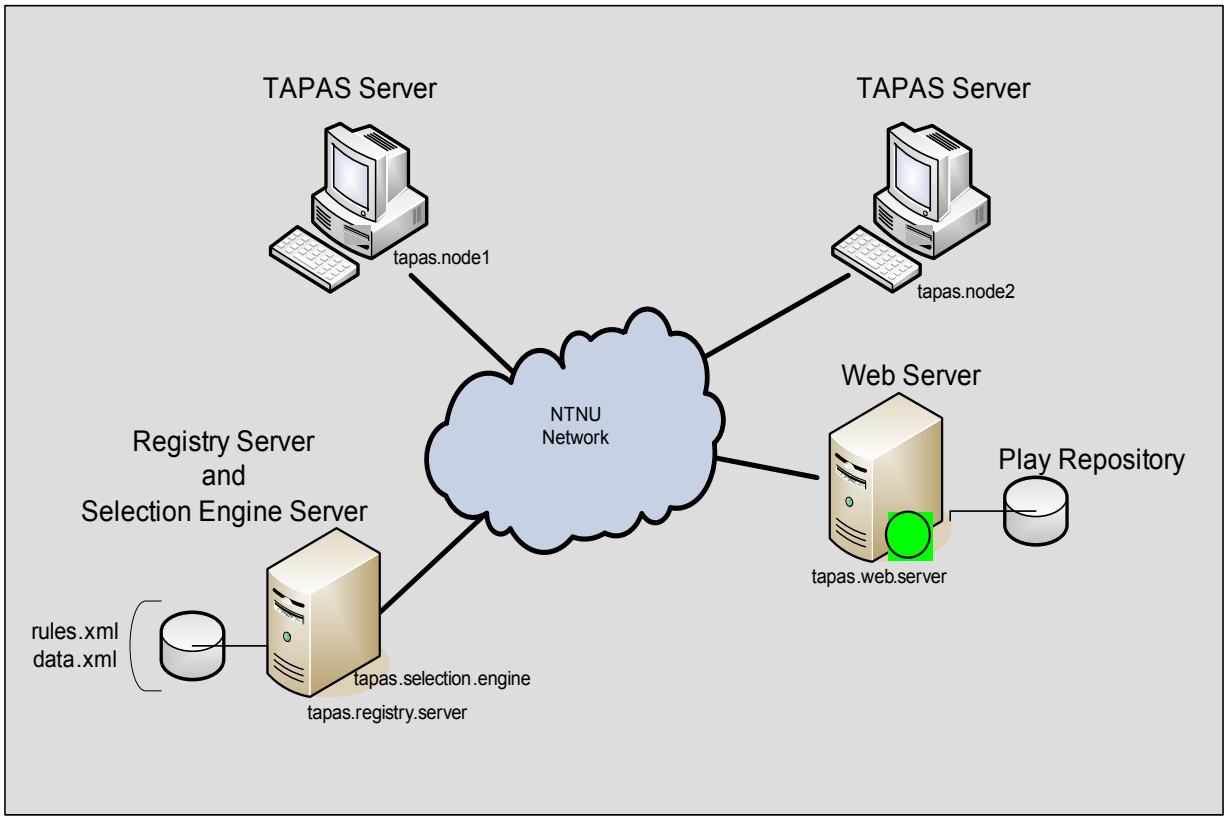


Figure 5-1: View of the experimentation scenario

- TAPAS Server: it is running a Web Server with SOAP support. In the demonstration this node is running the Apache Tomcat server as the Web server and the Apache Axis 1.1 as a SOAP aware server.
- Web Server: remote web server with access to a database which contains the code of the service that it is going to be plugged into the system. In this case, the Play Repository contains the manuscripts of the TeleSchool play and the TAPAS support system code.
- Registry Server: it is executed by the Java Registry Server included in the Java Web Services Developer Pack.
- Selection Engine Server: it is performed by a servlet running at the server. This servlet has access to the execution methods of the Selection Engine and to a database with the XML files needed by the Selection Engine.

This scenario is targeting at validating the feasibility of the implemented Web services architecture in TAPAS. The TAPAS service is described using WSDL and

the nodes offering this service can be discovered using the Registry Server. The communication between the TAPAS Server nodes is performed exchanging SOAP messages and the communication between the Director and the Selection Engine is performed by means of the SAAJ specification exchanging XML files attached to the SOAP messages.

5.2 Setting up the environment

In order to run the demonstration some settings are needed. The *TAPAS Server* nodes must be executing the TAPAS support system which is performed by the PNES. The PNES instance, with Web services support, needs a web server to handle the incoming SOAP messages and put into operation the endpoint of the offered service. The developed implementation in this thesis uses the Apache Tomcat as a Web server and the Apache Axis 1.1 as a SOAP server.

The Apache Ant build tool is used to compile the source code into the corresponding directories and to deploy the web service in the Apache Axis server. The file build.xml provided with the demonstration code contains the necessary *compile* and *deploy* tasks. The execution of these tasks is necessary before running the servers and starting the TAPAS support system.

The Tomcat server must be started before running the TAPAS support system; as the PNES instance needs its functionality for registering the service at the Registry Server. The Apache Axis is integrated into the Tomcat server, so both applications are started at the same time when the Tomcat server is started. From this moment, a WSDL description of the deployed service at the node is available at the endpoint address. This description can be used by any developer to create its own client application to interact with the offered service. In this example the WSDL description can be retrieved from each node using the internet address http://tapas.node1/axis/services/wsPNES?wsdl. The full WSDL description is shown in Appendix B.

The TAPAS support system needs some configuration properties in the start up process. An example of these configuration properties, which are specified in the confPlug-and-Play.properties file, is shown in Figure 5-2. This configuration file has

some improvements compared to the original configuration file used in TAPAS. For instance, comments and blank spaces are now allowed, i.e. providing an easy way to update and add more properties to the configuration file.

```
# The location of the codebase
codebase=http://tapas.web.server/WsPlug-and-Play/Plug-and-PlayRoot/

# homeinterface GAI
homeinterface=Actor://tapas.node1/pas1/WsPlug-and-Play.Director1

debugserver=localhost
nodeprofile=profDefault

# Local port where the local PNES is listening for local PAS instances requests
communicationport=9999

# URL of the Selection Engine Server. Only necessary at the Director's node.
xetServer=http://tapas.selection.engine/saaj-service/ServiceRequest

# Working directory where the query.xml and result.xml are located
dirWorking=C:/Program files/jwsdp-1.3/webapps/axis/WEB-INF/classes/files/

# Query's file names for each service available
School = tele-query.xml
Intellcom = intell-query.xml
```

Figure 5-2: Configuration file properties

Another element necessary for the demonstration which requires some settings is the Registry Server. The Java Registry Server that ships with the Java Web Services Developer Pack has been used for this purpose. Before using the Registry Server for the first time, it is necessary to publish the service taxonomy at the registry. This is done executing the *PublishScheme* Java application located at the *soaPlug-and-Playi* package. When the TAPAS service taxonomy is registered, a Universal Internet IDentifier (uiid) is then provided by the Registry Server. This uiid identifies uniquely in Internet the TAPAS service described by the WSDL description. The provided uiid is used on the publishing and discovery processes of the nodes that provide the service.

The Home Interface (HI) node, specified at the configuration properties file, is where the Director Role figure is created when a play is plugged in the repertory base. This node requires different settings than the other TAPAS Server nodes. The Director instance is the initiator of the Service Plug-In process and it is the only instance that communicates directly to the Selection Engine. It needs access to the XML query files that will be sent to the Selection Engine. The location of these XML query files is specified at the configuration properties file; the working directory property. In this demonstration only the *Initial Service Request* query is used and its content is shown in Appendix B.

The Selection Engine node comprises a servlet application with access to the XET-based selection engine. The start up of this node only requires a running web-server and the application servlet to be deployed at this server. The service endpoint of this servlet application is known by the Director node and it is specified at the configuration properties file.

Next section comprises the steps involved in the communication process with the Selection Engine and further execution of the calculated results. For a better comprehension of the process some fragments of the exchanged SOAP messages are shown with the explanation. The TCP Monitor of the Apache Axis has been used to capture the XML messages going back and forth between a SOAP client and server. To do this it is only necessary to run the TCP Monitor and configure the listening and destination ports. The monitor will receive the incoming requests in the listening port and redirect them to the real port where the service endpoint is listening. The start up of the TCP Monitor can be found at the Apache Axis user guide [26].

5.3 Demonstration

The experimentation scenario has been run using some features of the Dynamic Service Management framework under the TAPAS support platform. According to the Dynamic Service Management framework [13], the available requests specified in XML are Initial Service request, Role-Figure move and Function Update request. These requests are processed by the Selection Engine and the results are a *Role Figure Specification* and a *Mapping table.* This calculated *Role Figure Specification* and *Mapping table* are then sent to the proper State Machine Interpreter (SMI) to

instantiate it and then execute it. The SMI is the primary entity in the framework responsible for the execution of Role-Figures according to instantiated Role-Figure Specifications sent by the Service Manager. It is assumed that every node executing Role-Figures is running a SMI, and is characterized by a set of capabilities [13].

In this demonstration the Director Role-Figure will send an *Initial Service* request to the Selection Engine. This request specifies the Role-Figure to be plugged and the node that will perform it. The *Role Figure Specification* and the *Mapping table* are sent back to the Director who will redirect them to the specified node at the Initial Service Request. The steps followed in this communication process are described in the next sections: Publish the TAPAS Server node, Plug-in the TeleSchool play and Plug-in the TeleSchool service using the *Initial Service* request.

5.3.1 Register the TAPAS Server node

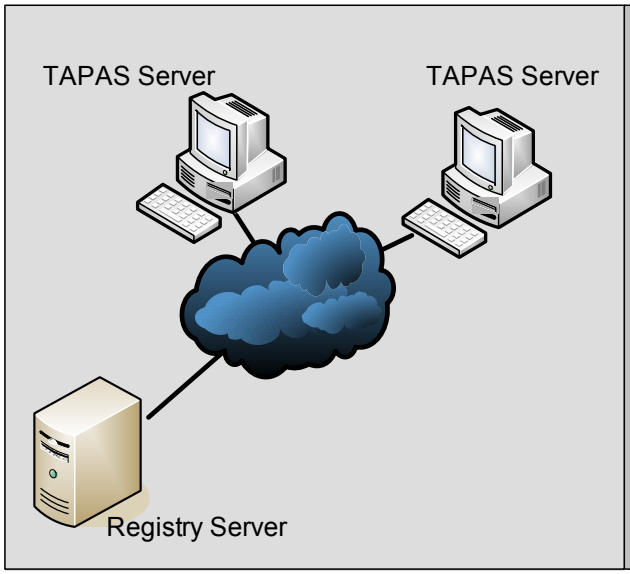
At the start up of the TAPAS support system, the PNES instance registers itself automatically into the Registry Server using the uiid provided in the tapasconcepts.xml file. The code which performs this task is in the soaPlug-and-Playi.RegistryServer Java class. And the configurable properties used by this application are at the registryprops.properties file. These properties specify the address of the Registry Server, identification data of the PNES willing to register and the information required by the Registry Server to register a new active node offering the TAPAS service. An example of the registry connection properties used at the configuration file is shown in Figure 5-3.

```
##Registry Specific properties

query.url=http://tapas.registry.server/RegistryServer/
publish.url=http://tapas.registry.server/RegistryServer/
user.name=testuser
user.password=testuser

##if you are behind a firewall this needs to be configured
http.proxy.host=
http.proxy.port=
```

```
## Values used for the publish process at the start up of the TAPAS support system
## at each node
org.name=NTNU
org.description=TAPAS project
person.name=Telematics
phone.number=
email.address=

classification.scheme=TAPASScheme
classification.name=TAPASServices
classification.value=wsPNES
service.name=wsPNES
service.description=TAPAS WebServices PNES support
svcbinding.description=wsPNES ServiceBinding
svcbinding.accessURI=http://tapas.node1/axis/services/wsPNES

## Specific values used by the PublishScheme Java class to register the wsPNES
## service taxonomy in the Registry Server

tapas.scheme.name=TAPASScheme
tapas.scheme.description=A ClassificationScheme for TAPAS
tapas.classification.name=TAPASServices
tapas.classification.value=TAPASServices
tapas.scheme.link=http://129.241.209.39:8080/axis/services/wsPNES?wsdl
tapas.scheme.linkdesc=wsPNES Web Service Description
```

Figure 5-3: Example of the registryprops.properties configuration file.

The two TAPAS server nodes considered at the demonstration scenario are registered at the Registry Server. It is assumed that both nodes are running the Apache Axis server which will provide the WSDL description of the service. The WSDL description is validated by the Registry Server at the registration process. Figure 5-4 illustrates an overview of the registering process and the involved elements.



When the TAPAS service is started at a node , the PNES instance is registered in the database of the Registry Server . This registration process is done according to the UDDI specification .

The communication process is synchronous and uses the connection properties established in the registryprops .properties file .

The tapasconcepts .xml file which contains the uiid of the TAPAS service and the tapasconcepst .dtd file are needed to associate the node to the correct service . This ensures that the offered service is the TAPAS service and validates its WSDL description .

Figure 5-4: Publishing the TAPAS Server node at the start up in the Registry.

5.3.2 Plug-in the TeleSchool play

In this step, the TeleSchool manuscript is put into the repertory base. The PlayPlugIn process can be started by any node running the TAPAS support system at the network. The PlayPlugIn command includes the location of the Web-server that contains the manuscripts. The command is inserted at the command line window of the PNES instance, this is done in the same way as it is done when running the original TAPAS system. The TeleSchool play manuscript is downloaded from the Web-server to the repertory base. This action comprises the instantiation of a Director Role figure if it does not exist. The Director is instantiated at the Home Interface node, which specified at the configuration properties file. The Actor Support layer is needed to instantiate the Director Role Figure. If the actor support is not running at the node a PAS instance will be started. As a result of this action the PAS is started at the node specified in the configuration file as Home Interface. The Director Role figure is also created if none existed before. The whole process is illustrated in Figure 5-5.

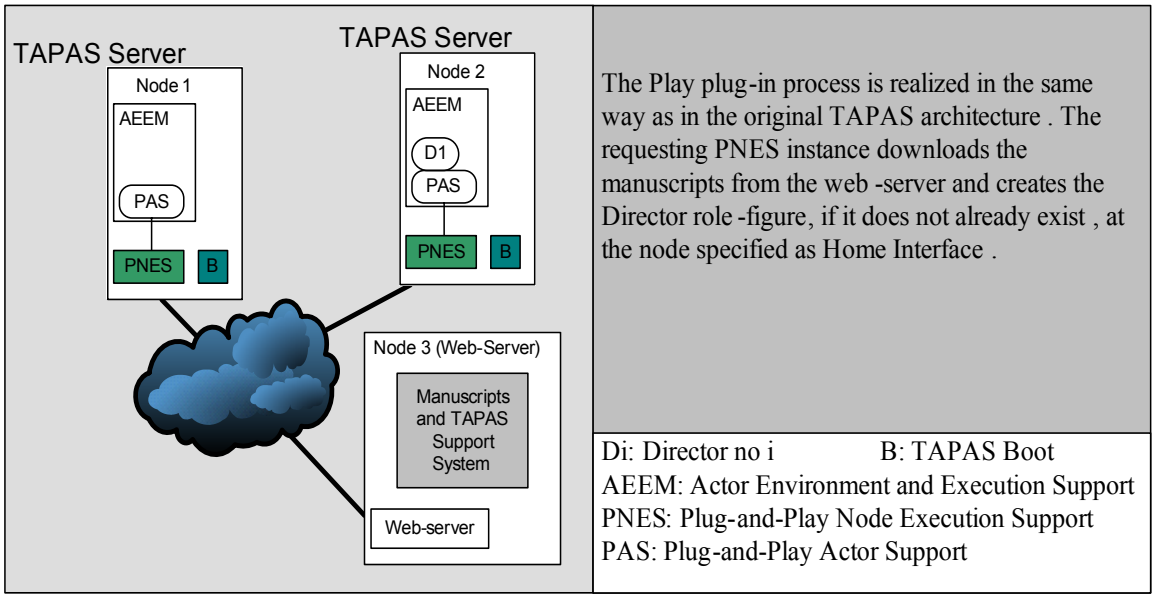
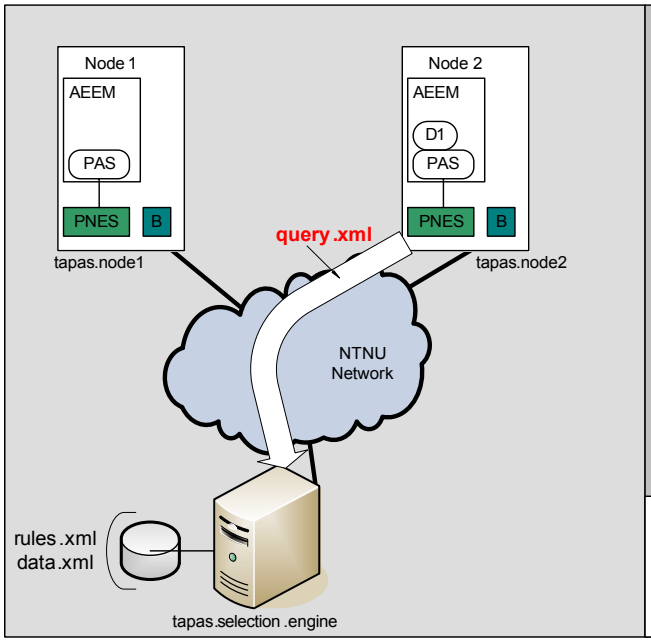


Figure 5-5: TeleSchool Play Plug-in.

5.3.3 Plug-in the TeleSchool service

The Service plug-in process is the fundamental part of this demonstration. In this stage the application integration is presented. The TAPAS support system is developed with Java and the Selection Engine, the remote application we want to use consumes XML documents. The integration problem, as reported throughout this thesis, has been solved with the implementation of Web services architecture in TAPAS.

The Service plug-in process has been divided into four snapshots, each one associated to one of the messages exchanged in the plug-in procedure. The illustrations are followed by the exchanged SOAP message in the communication process. These messages have been captured using the TCP Monitor provided with the Apache Axis. The steps involved in this process are: sending the *Initial Service Request* query to the Selection Engine (Figure 5-6), the Director processes the response (Figure 5-8), ensuring that the receiver node exists (Figure 5-10), sending the ActorPlugIn request to the destination node (Figure 5-11).



The Director role figure is the initiator of the service plug-in request. In the first step of the plug-in process the Director sends the *Initial Service Request* xml query attached to the SOAP message using the SAAJ specification. The query is sent to the Selection Engine using a client application.

The Selection Engine node is executing a servlet application which extracts the xml query from the request message. Then, this query is processed by the Selection Engine.

Di: Director no i B: TAPAS Boot
AEEM: Actor Environment and Execution Support
PNES: Plug-and-Play Node Execution Support
PAS: Plug-and-Play Actor Support

Figure 5-6: Sending the *Initial Service Request* query to the Selection Engine.

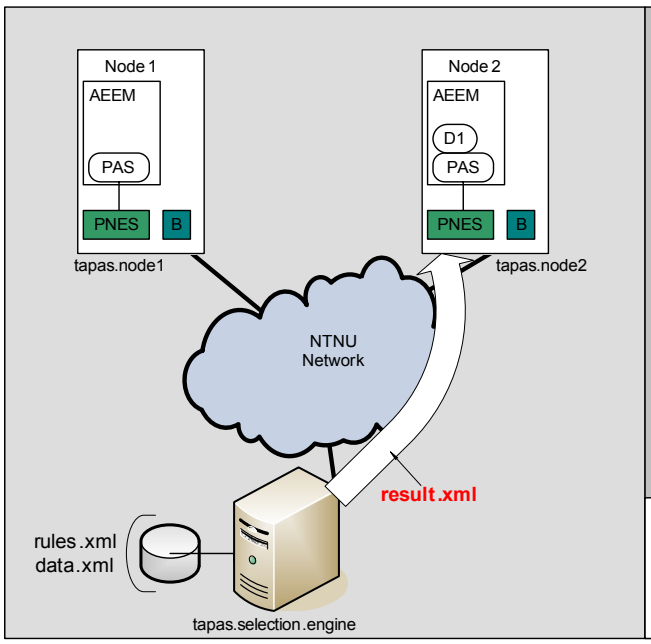
The SOAP message carries the query.xml file attached to the message. This attachment part is compliant to the SAAJ specification and represents the *Initial Service Request*. The message captured by the TCP Monitor is shown in Figure 5-7.

```
POST /saaj-service/ServiceRequest HTTP/1.1
Content-Type: multipart/related; type="text/xml";
boundary="----=_Part_1_8344960.1083145695955"
Content-Length: 981
SOAPAction: ""\par
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Java/1.4.2_01
Host: 127.0.0.1\par
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
------=_Part_1_8344960.1083145695955
Content-Type: text/xml

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:rdf="urn:org.tapas.rdf" xmlns:tns="urn:org.tapas">
  <SOAP-ENV:Body>
    <tns:Intelcom/>\
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
------=_Part_1_8344960.1083145695955Content-Type: text/xml
<InitialServiceRequest type="InitialServiceRequest">
  <sender />
  <dateTime />
  <serviceType>TeleSchool</serviceType>
  <roleRequesting>SchoolClient</roleRequesting>
  <preferredConfiguration>
    <nodeInstalling>http://comp1.tapas.org</nodeInstalling>
  </preferredConfiguration>
  <contextInfo>
    <connectionUsed>Bluetooth</connectionUsed>
    <userSubscription>Advanced</userSubscription>
    <MMSupport>Speaker</MMSupport>
  </contextInfo>
  <Result>\
    <Manus>Svar_MName</Manus>
    <ActionGroup>Svar_Gi</ActionGroup>
    <Category>Svar_CapCategory</Category>
  </Result>
</InitialServiceRequest>
------=_Part_1_8344960.1083145695955--
```

Figure 5-7: SOAP message with attached *Initial Service Request* XML file.

After receiving the query, the Selection Engine processes it and calculates the Role-Figure Specification and the Mapping table as it was reported in section 2.3.2 of this thesis. Figure 5-8 shows the Selection Engine server node sending back the SOAP response to the Director node.



The Selection Engine processes the received query using the rules.xml and the data.xml files from the database. The result.xml file contains the calculated *Role-Figure Specification* and the *Mapping table*. The xml result is attached to the SOAP response message and it is sent back to the Director.

Di: Director no i B: TAPAS Boot
AEEM: Actor Environment and Execution Support
PNES: Plug-and-Play Node Execution Support
PAS: Plug-and-Play Actor Support

Figure 5-8: Sending to the Director the calculated *Role-Figure Specification* and *Mapping table*.

The SOAP response is presented in Figure 5-9. This response contains the calculated Role-Figure Specification and the Mapping table. These results are provided by the Selection Engine in a XML file, the result.xml. This file is attached to the SOAP message following the SAAJ specification.

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.4
Set-Cookie: JSESSIONID=8BE643B5A3D29407CDDA77C431EFB2ED;
Path=/saaj-service
SOAPAction: ""
Content-Type: multipart/related; type="text/xml";
boundary="----=_Part_1_1340668.1083145700439"
Content-Length: 4666
Date: Wed, 28 Apr 2004 09:48:20 GMT
Server: Sun-Java-System/JWSDP-1.3
------=_Part_1_1340668.1083145700439
Content-Type: text/xml
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:rdf="urn:org.tapas.rdf"
xmlns:tns="urn:org.tapas" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <tns:Intelcom/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

------=_Part_1_1340668.1083145700439Content-Type: text/xml
<Answer>
  <InitialServiceRequest type="InitialServiceRequest">
    <sender/>
    <dateTime/>
    <serviceType>     TeleSchool    </serviceType>
    <roleRequesting>     SchoolClient    </roleRequesting>
    <preferredConfiguration>
      <nodeInstalling>       http://comp1.tapas.org      </nodeInstalling>
    </preferredConfiguration>
    <contextInfo>
      <connectionUsed>       Bluetooth      </connectionUsed>
      <userSubscription>       Advanced      </userSubscription>
      <MMSupport>       Speaker      </MMSupport>
    </contextInfo>
    <Result>
      <Manus>       SchoolClient_Advanced      </Manus>
      <ActionGroup>       G5      </ActionGroup>
      <Category>       C30      </Category>
    </Result>
  </InitialServiceRequest>

  <InitialServiceRequest type="InitialServiceRequest">
    <sender/>
    <dateTime/>
<serviceType>     TeleSchool    </serviceType>
    <roleRequesting>     SchoolClient    </roleRequesting>
    <preferredConfiguration>
      <nodeInstalling>       http://comp1.tapas.org      </nodeInstalling>
    </preferredConfiguration>
```

```
    <contextInfo>
      <connectionUsed>      Bluetooth     </connectionUsed>
      <userSubscription>      Advanced     </userSubscription>
      <MMSupport>      Speaker     </MMSupport>
    </contextInfo>
    <Result>
      <Manus>      SchoolClient_Advanced     </Manus>
      <ActionGroup>      G4     </ActionGroup>
      <Category>      C21     </Category>
    </Result>
  </InitialServiceRequest>
  <InitialServiceRequest type="InitialServiceRequest">
    <sender/>
    <dateTime/>
    <serviceType>     TeleSchool    </serviceType>
    <roleRequesting>     SchoolClient    </roleRequesting>
    <preferredConfiguration>
      <nodeInstalling>      http://comp1.tapas.org     </nodeInstalling>
    </preferredConfiguration>
    <contextInfo>
      <connectionUsed>      Bluetooth     </connectionUsed>
      <userSubscription>      Advanced     </userSubscription>
      <MMSupport>      Speaker     </MMSupport>
    </contextInfo>
<Result>
      <Manus>      SchoolClient_Advanced     </Manus>
      <ActionGroup>      G2     </ActionGroup>
      <Category>      C10     </Category>
    </Result>
  </InitialServiceRequest>
  <InitialServiceRequest type="InitialServiceRequest">
    <sender/>
    <dateTime/>
    <serviceType>     TeleSchool    </serviceType>
    <roleRequesting>     SchoolClient    </roleRequesting>
    <preferredConfiguration>
      <nodeInstalling>      http://comp1.tapas.org     </nodeInstalling>
    </preferredConfiguration>
    <contextInfo>
      <connectionUsed>      Bluetooth     </connectionUsed>
      <userSubscription>      Advanced     </userSubscription>
      <MMSupport>      Speaker     </MMSupport>
    </contextInfo>
    <Result>
      <Manus>      SchoolClient_Advanced     </Manus>
      <ActionGroup>      G3     </ActionGroup>
      <Category>      C1     </Category>
    </Result>
</InitialServiceRequest>
  <InitialServiceRequest type="InitialServiceRequest">
    <sender/>
    <dateTime/>
    <serviceType>     TeleSchool    </serviceType>
    <roleRequesting>     SchoolClient    </roleRequesting>
    <preferredConfiguration>
      <nodeInstalling>      http://comp1.tapas.org
    </nodeInstalling>
    </preferredConfiguration>
```

```
        <contextInfo>
        <connectionUsed>      Bluetooth     </connectionUsed>
        <userSubscription>      Advanced     </userSubscription>
        <MMSupport>      Speaker     </MMSupport>
      </contextInfo>
      <Result>
        <Manus>      SchoolClient_Advanced     </Manus>
        <ActionGroup>      G1     </ActionGroup>
        <Category>      C1     </Category>
      </Result>
    </InitialServiceRequest>
</Answer>
------=_Part_1_1340668.1083145700439—
```

Figure 5-9: SOAP message response with attached XML result file.

The next step, once the result.xml file has been read by the Director, is to execute the requested query. The *Initial Service Request* comprises the plug-in of a SchoolClient Role figure at a specified node. But before sending the ActorPlugIn request to the destination node, the PNES should ensure that this is an active TAPAS node. This is shown in Figure 5-10, the PNES instance queries the Registry Server in order to *discover* the registered TAPAS Server nodes. Then, it looks for the destination node at the received result. If the information regarding the destination node is correct and the TAPAS service is operative, the ActorPlugIn request will be sent. Otherwise the service plug-in is cancelled and a message error is shown.

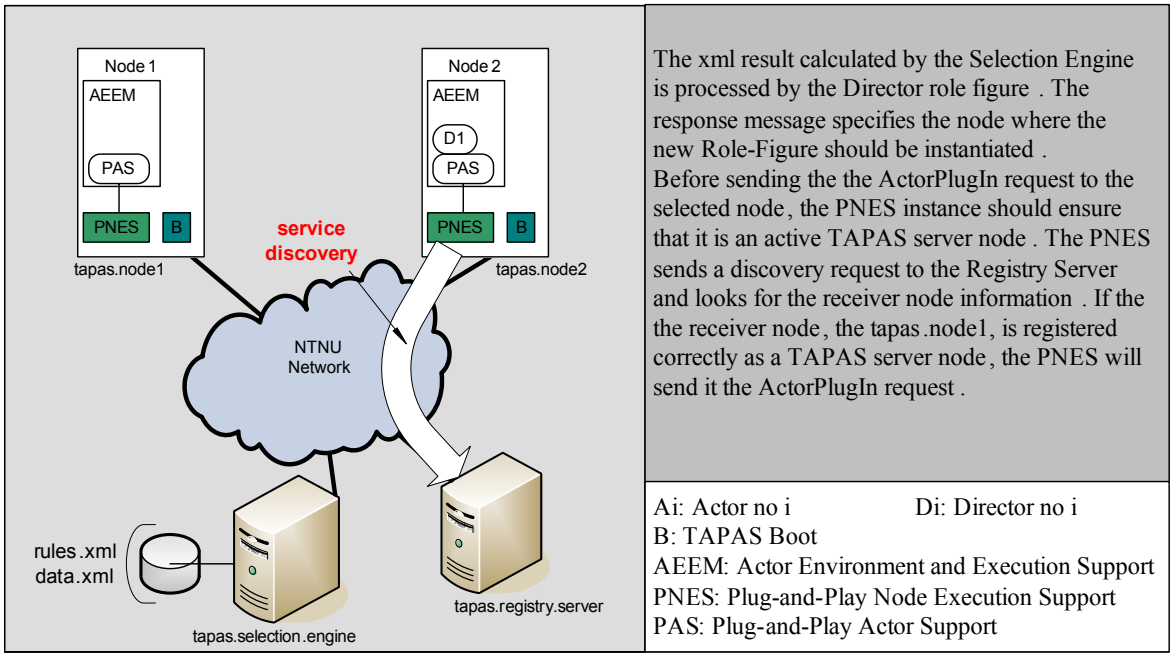


Figure 5-10: Discovery and identification of the destination TAPAS Server node.

Figure 5-11 illustrates the last step in the service plug-in process. The ActorPlugIn request encoded in a SOAP message is sent to the destination node. The parameters of this request include the *Mapping table* calculated at the Selection Engine. All parameters are serialized from Java data types to standard types. These standard data types should be supported by any XML-based platform. Any developer, under any platform, can create its own client application able to understand the parameters and the request logic sent encoded in the SOAP message. This feature renders possible the implemented Web-services architecture in TAPAS to interoperate with other non-Java platforms, thus offering a feasible solution to the application integration within the TAPAS platform.

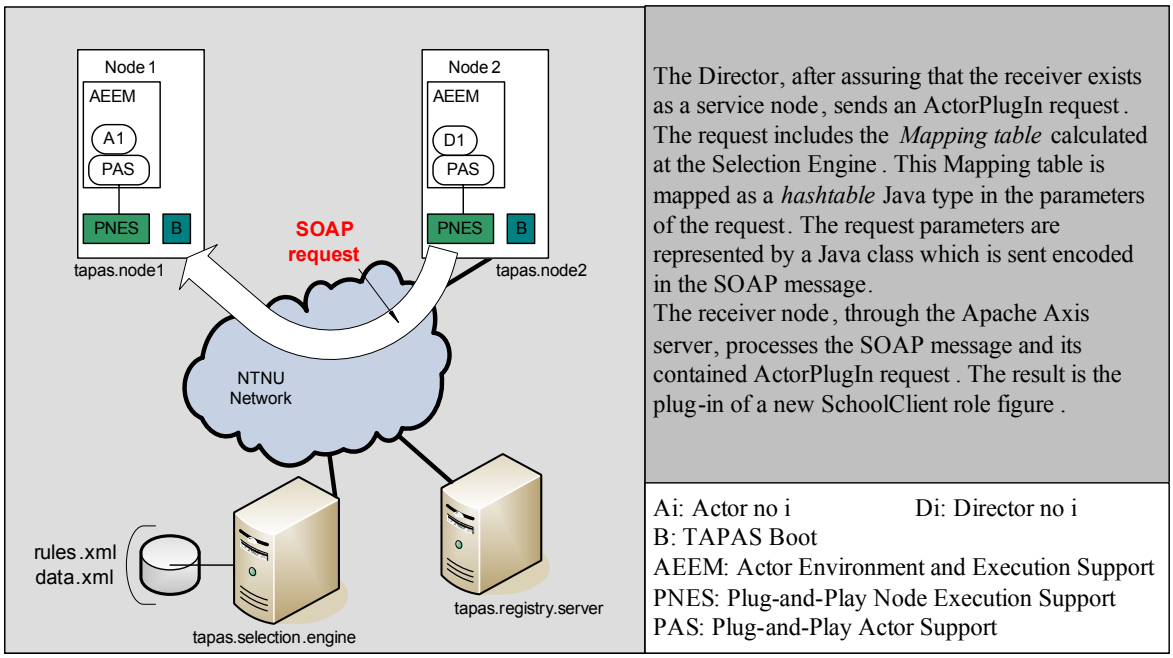


Figure 5-11: Sending the ActorPlugIn request encoded in a SOAP message.

The SOAP message, shown in Figure 5-12, is sent from the Director to the destination node where the new SchoolClient Role figure will be instantiated. The SOAP message encodes the parameters of the ActorPlugIn request as it is described in the WSDL description of the service. Only the parameters are sent with the message. The service endpoint, supported by the Apache Axis server, will extract the request parameters and it will execute the appropriated methods at the local PNES instance.

```
POST /axis/services/wsPNES HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.1
Host: 127.0.0.1
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 8663
<?xml version="1.0" encoding="UTF-8"?>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <ns1:syncRequestFromPNES
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="http://WsPlug-
and-Play">
        <in0 href="#id0"/>
      </ns1:syncRequestFromPNES>
      <multiRef id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:RequestPars"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns2="http://WsPlug-and-Play">
        <mapTable href="#id1"/>
        <requestType xsi:type="xsd:int">4</requestType>
        <sender href="#id2"/>
        <receiver href="#id3"/>
        <PlayPlugIn xsi:type="xsd:int">1</PlayPlugIn>
        <PlayChangesPlugIn xsi:type="xsd:int">2</PlayChangesPlugIn>
        <PlayPlugOut xsi:type="xsd:int">3</PlayPlugOut>
        <ActorPlugIn xsi:type="xsd:int">4</ActorPlugIn>
        <ActorPlugOut xsi:type="xsd:int">5</ActorPlugOut>
        <ActorBehaviourPlugIn xsi:type="xsd:int">6</ActorBehaviourPlugIn>
        <ActorBehaviourPlugOut xsi:type="xsd:int">8</ActorBehaviourPlugOut>
        <ActorChangeBehaviour xsi:type="xsd:int">7</ActorChangeBehaviour>
        <ActorPlay xsi:type="xsd:int">9</ActorPlay>
        <subscribeRequest xsi:type="ns2:SubscribeRequest" xsi:nil="true"/>
        <subscribeReport xsi:type="ns3:ArrayOf_xsd_string" xsi:nil="true"
xmlns:ns3="http://xml.apache.org/xml-soap"/>
        <subscribeCancel xsi:type="xsd:string" xsi:nil="true"/>
        <RoleSessionAction xsi:type="xsd:int">13</RoleSessionAction>
        <ActorCapabilities xsi:type="xsd:int">14</ActorCapabilities>
        <RT xsi:type="soapenc:Array" soapenc:arrayType="xsd:string[15]"
xmlns:ns4="http://xml.apache.org/xml-soap">
          <item>none</item>
          <item>PlayPlugIn</item>
          <item>PlayChangesPlugIn</item>
          <item>PlayPlugOut</item>
          <item>ActorPlugIn</item>
          <item>ActorPlugOut</item>
          <item>ActorBehaviourPlugIn</item>
          <item>ActorChangeBehaviour</item>
          <item>ActorBehaviourPlaugOut</item>
          <item>ActorPlay</item>
          <item>SubscribeRequest</item>
          <item>SubscribeReport</item>
          <item>SubscribeCancel</item>
          <item>RoleSessionAction</item>
          <item>ActorCapabilities</item>
        </RT>
        <play xsi:type="ns2:Play" xsi:nil="true"/>
```

```
<actorPlugInReq href="#id4"/>
        <plugOutRoleSession xsi:type="ns2:RoleSession" xsi:nil="true"/>
        <plugOutActor xsi:type="ns2:GAI" xsi:nil="true"/>
        <apo xsi:type="xsd:boolean">false</apo>
        <upgradePars xsi:type="ns5:ArrayOf_xsd_string" xsi:nil="true"
xmlns:ns5="http://xml.apache.org/xml-soap"/>
        <applicationMessage xsi:type="ns2:ApplicationMessage" xsi:nil="true"/>
        <roleSession href="#id5"/>
        <capOpType xsi:type="xsd:int">0</capOpType>
        <capabilities xsi:type="ns2:CapabilitySet" xsi:nil="true"/>
      </multiRef>
      <multiRef id="id5" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns6:RoleSession"
xmlns:ns6="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        <roleSessionId xsi:type="xsd:string">RS://129.241.209.39/pas1/WsPlug-and-
Play.Director1/0</roleSessionId>
        <initiator href="#id2"/>
        <cooperator href="#id3"/>
      </multiRef>
      <multiRef id="id2" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns7:GAI"
xmlns:ns7="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        <type xsi:type="xsd:string">Actor</type>
        <node xsi:type="xsd:string">129.241.209.39</node>
        <address xsi:type="xsd:string">129.241.209.39</address>
        <PAS xsi:type="xsd:string">pas1</PAS>
        <name xsi:type="xsd:string">WsPlug-and-Play.Director1</name>
        <hmhandles href="#id6"/>
        <initialized xsi:type="xsd:boolean">true</initialized>
      </multiRef>
      <multiRef id="id1" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns8:Map"
xmlns:ns8="http://xml.apache.org/xml-soap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        <item>
          <key xsi:type="xsd:string">G4</key>
          <value xsi:type="xsd:string">C21</value>
        </item>
        <item>
          <key xsi:type="xsd:string">G2</key>
          <value xsi:type="xsd:string">C10</value>
        </item>
        <item>
          <key xsi:type="xsd:string">G5</key>
          <value xsi:type="xsd:string">C30</value>
        </item>
        <item>
          <key xsi:type="xsd:string">G3</key>
          <value xsi:type="xsd:string">C1</value>
        </item>
        <item>
          <key xsi:type="xsd:string">G1</key>
          <value xsi:type="xsd:string">C1</value>
        </item>
      </multiRef>
      <multiRef id="id3" soapenc:root="0"
```

```
<type xsi:type="xsd:string">Actor</type>
          <node xsi:type="xsd:string">129.241.208.154</node>
          <address xsi:type="xsd:string">129.241.208.154</address>
          <PAS xsi:type="xsd:string">pas1</PAS>
          <name xsi:type="xsd:string">SchoolClient</name>
          <hmhandles href="#id6"/>
          <initialized xsi:type="xsd:boolean">true</initialized>
        </multiRef>
        <multiRef id="id4" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns10:ActorPlugInReq" xmlns:ns10="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <location href="#id7"/>
          <role href="#id8"/>
          <play href="#id9"/>
          <rqCaps href="#id10"/>
          <rsCaps href="#id11"/>
        </multiRef>
        <multiRef id="id6" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns11:Map"
xmlns:ns11="http://xml.apache.org/xml-soap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <item>
            <key xsi:type="xsd:string">PNES://129.241.208.154/pas1/pas1</key>
            <value xsi:type="xsd:string">http://129.241.208.154:8080/axis/services/wsPNES</value>
          </item>
        </multiRef>
        <multiRef id="id9" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns12:Play"
xmlns:ns12="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <playId xsi:type="xsd:string">School</playId>
          <playVer xsi:type="xsd:string">v1_1</playVer>
          <playLoc xsi:type="xsd:string">http://www.stud.ntnu.no/~vilaarme/WsPlug-and-Play/Plug-
and-PlayRoot/</playLoc>
          <verA xsi:type="xsd:int">0</verA>
          <verB xsi:type="xsd:int">0</verB>
        </multiRef>
        <multiRef id="id8" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns13:Role"
xmlns:ns13="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <roleId xsi:type="xsd:string">SchoolClient</roleId>
        </multiRef>
        <multiRef id="id7" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns14:GAI"
xmlns:ns14="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <type xsi:type="xsd:string">Actor</type>
          <node xsi:type="xsd:string">129.241.208.154</node>
          <address xsi:type="xsd:string">129.241.208.154</address>
          <PAS xsi:type="xsd:string">pas1</PAS>
          <name xsi:type="xsd:string">SchoolClient</name>
          <hmhandles href="#id6"/>
          <initialized xsi:type="xsd:boolean">true</initialized>
        </multiRef>
        <multiRef id="id11" soapenc:root="0"
          soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
<capabilities href="#id12"/>
          <Add xsi:type="xsd:int">2</Add>
          <Set xsi:type="xsd:int">1</Set>
          <Remove xsi:type="xsd:int">3</Remove>
          <all xsi:type="ns16:ArrayOf_xsd_string" xsi:nil="true"
xmlns:ns16="http://xml.apache.org/xml-soap"/>
        </multiRef>

        <multiRef id="id10" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xsi:type="ns17:CapabilitySet" xmlns:ns17="http://WsPlug-and-Play"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          <capabilities href="#id13"/>
          <Add xsi:type="xsd:int">2</Add>
          <Set xsi:type="xsd:int">1</Set>
          <Remove xsi:type="xsd:int">3</Remove>
          <all xsi:type="ns18:ArrayOf_xsd_string" xsi:nil="true"
xmlns:ns18="http://xml.apache.org/xml-soap"/>
        </multiRef>
        <multiRef id="id12" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns19:Vector"
xmlns:ns19="http://xml.apache.org/xml-soap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"/>
        <multiRef id="id13" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns20:Vector"
xmlns:ns20="http://xml.apache.org/xml-soap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"/>
      </soapenv:Body>
    </soapenv:Envelope>
```

Figure 5-12: SOAP message that encodes the ActorPlugIn request.

6 Conclusion and further work

The purpose of this thesis was to provide a viable solution to the application integration issue present in the TAPAS platform. Therefore it enables the interaction with non-Java platforms and achieves a cross-platform distributed system. Furthermore, it was required to integrate the XET-based Selection Engine of the TAPAS Dynamic Configuration architecture into the TAPAS core platform. The solution proposed in this thesis is the implementation of Web services architecture in TAPS. In addition, a demonstration using an existing TAPAS application was needed to show the feasibility of the implemented Web service architecture in TAPAS.

The communication infrastructure of the TAPAS architecture has been redesigned. The communication between nodes running the TAPAS service is done exchanging SOAP messages. The parameters of the TAPAS requests are serialized into standard data types understandable by any Web compatible platform. Thus, this allows any client to interact with the TAPAS service deployed at the node. The Registry is based on the UDDI service, and Web services technology is used for discovery and upgrade procedures with the Registry.

The integration with the Selection Engine has been achieved. The TAPAS platform is able to send XML queries to the Selection Engine for its process. The calculated result, also in XML format, is consumed by the TAPAS node and the required actions are executed.

The TeleSchool TAPAS application is used as the scenario for the demonstration. The demonstration shows examples of the SOAP messages sent back and forth between the client and the server application. Upon a request of the Director, a SchoolClient Role figure is instantiated at the corresponding node with the results calculated by the Selection Engine.

This implementation has been proved to be valid for the application integration issue of this thesis. However, it was also desired a lightweight support platform with a simple communication protocol and faster than the RMI implementation used in the TAPAS core platform. The implemented communication infrastructure with Web

services support presents extra run-time checks, and the text-based data used in XML makes it inefficient. So the TAPAS application using Web services is several times slower than the implemented using binary data like Java RMI. In addition, sending plain-text XML across the open Internet makes it vulnerable to security breaches. Therefore more work is needed in the security field to overcome the lacks of the Web services implementation.

At this time, some thesis and projects are working on XML specifications of the manuscripts. The manuscripts and the Role-figures specifications used for this thesis were implemented in Java. This adds extra time in the communication process as the parameters have to be serialized and deserialized at the endpoints. Working with XML specifications of the Role figures and XML manuscripts will simplify the communication process. So, further work has to be done in this area to achieve an all-XML based architecture for the TAPAS platform.

References

1. Anne Thomas Manes, "*Web Services: A Manager's Guide"*, Addison-Wesley Information Technology Series, June 2003.

2. Keith Ballinger, ".NET Web Services: Architecture and Implementation", Addison-Wesley, February 2003.

3. Walsh, Aaron E., "UDDI, SOAP, and WSDL: The Web Services Specification Reference Book", Prentice Hall PTR, 2002.

4. Aagesen, F. A., Helvik, B. E. Wuwongse, V., Meling, H., Bræk, R., Johansen, U. (1999) *Towards a Plug and Play Architecture for Telecommunications*. IFIP Fifth International Conference on Intelligence in Networks (SmartNet99), Bankok – Thailand. Available online: http://tapas.item.ntnu.no/publication/smartnet99.pdf

5. Finn Arve Aagesen, Bjarne Helvik, Ulrik Johansen and Hein Meling. (2001) *Plug and Play for telecommunication functionality – architecture and demonstration issues*, The International Conference on Information Technology for the New Millennium (IConIT2001), Thammasat University, Bangkok - Thailand, May 2001.

6. Aagesen, F. A., Anutariya, C., Shiaa, M. M. and Helvik, B. E. (2002) *Support Specification and Selection in TAPAS*. Proc. IFIP WG6.7 Workshop and EUNICE Summer School on Adaptable Networks and Teleservices, Trondheim, Norway, September, 109-116.

7. Ulrik Johansen, "Dynamic Plug and Play - What is it, what are the advantages of using it?" presented at IT-PRO 2000, Sandefjord, Norway.

8. Mazen Malek Shiaa and Finn Arve Aagesen, "*Mobility management in a Plug and Play architecture*", IFIP TC6 Seventh International Conference on Intelligence in Networks, Saariselka, Finland, April 2003. Published by Kluwer Academic Publishers.

9. Mazen Malek Shiaa and Lars Erik Liljeback, "*User and Session Mobility in a Plug-and-Play Network Architecture*", IFIP WG6.7 Workshop and EUNICE Summer School on Adaptable Networks and Teleservices, Trondheim -Norway, September 2002.

10. Mazen Malek Shiaa and Finn Arve Aagesen, "Architectural Considerations for Personal Mobility In the Wireless Internet", Personal Wireless Communication (PWC2002), Singapore, October 2002.

11. Aagesen, F. A., Anutariya, C., Shiaa, M. M., Helvik, B. E. (2002). *Support Specification and Selection in TAPAS*. IFIP WG6.7 Workshop and Eunice Summer School on Adaptable Networks and Teleservices, September 2002, Trondheim – Norway. [http://tapas.item.ntnu.no/publication/euniceCap2002.pdf]

12. Aagesen, F. A., Helvik, B. E., Anutariya, C., and Shiaa M. M. (2003) On Adaptable Networking, *Proc. 2003 Int'l Conf. Information and Communication Technologies (ICT 2003)*, Thailand.

13. Mazen Malek Shiaa, Shanshan Jiang, Paramai Supadulchai and Joan J. Vila-Armengol (2004). *An XML-based Framework for Dynamic Service Management.* The 2004 IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM 04), *23 - 26 November 2004, Bangkok, Thailand. [Submitted]*

14. Aagesen, F. A., Anatariya, C., Shiaa, M., M., Helvik, B. E. (2003). *Dynamic Configuration of Plug-and-Play Systems.* NOMS 2004, Korea.

15. Johansen, U., Aagesen, F. A., Helvik, B. E., Meling, H. (1999). *Demonstrator – Requirements and functional description.* Plug-and-Play Technical Report, Department of Telematics, NTNU, ISSN 1500-3868

16. Wuwongse, V., Anutariya, C., Akama, K. and Nantajeewarawat, E. (2001) XML Declarative Description (XDD): A Language for the Semantic Web. IEEE Intelligent Systems 16(3): 54–65.

17. Inger Anne Tøndel, Dynamic Configuration of Plug-and-Play Systems, Project Report, Department of Telematics, NTNU, 2003. Available online at http://tapas.item.ntnu.no/publications/IngerAnnP.doc .

18. Eirik Lühr, Mobility support for wireless devices - within the TAPAS platform, MSc thesis, Department of Telematics, NTNU, 2004. Available online at http://tapas.item.ntnu.no/publications/EirikTh.pdf .

19. Kim Topley, "Java Web Services in a Nutshell", O'Reilly, June 2003.

20. Dion Almaer, "Creating Web Services with Apache Axis", May 2002, http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html [Accessed February 2004]

21. Robert Englander, "Java and SOAP", O'Reilly, May 2002, Chapter 5: Working with Complex Data Types.

22. Brett McLaughlin, “Java and XML, 2nd Edition”, O’Reilly, September 2001, Chapter 12: SOAP. Available online: O’Reilly Book Excerpts http://www.onjava.com/pub/a/onjava/excerpt/java_xml_2_ch2/index.html [Accessed February 2004]

23. Robert Hustead, “Mapping XML to Java, Part 1”, http://www.javaworld.com/javaworld/jw-08-2000/jw-0804-sax.html [Accessed March 2004]

24. Dennis M. Sosnoski, “XML documents on the run, Part 1”, February 2002, http://www.javaworld.com/javaworld/jw-02-2002/jw-0208-xmljava.html [Accessed March 2004]

25. SUN Microsystems, the Web services Homepage, Java Web Services Developer Pack (WSDP) documentation, http://java.sun.com/webservices/index.jsp [Accessed June 2004]

26. The Apache homepage, Apache Axis 1_1, http://ws.apache.org/axis/ [Accessed June 2004]

27. KR laboratory, *XET Engine homepage*, http://kr.cs.ait.ac.th/xet/ [Accessed June 2004]

28. TAPAS, website. Available online: http://tapas.item.ntnu.no [Accessed June 2004]

29. Castro-Leon, Enrique, The Web within the Web, *IEEE Spectrum Magazine,* pp. 36-40, February 2004.

Appendix A: Overview of Web Services

XML is a universal standard for representing data, so XML-based programs are inherently interoperable. Basically, XML uses the lowest common data denominator available, which is text. Here is how it works: data in XML form is consigned to specific fields. There might be one field for "price," for example, and another for "quantity." Once information is in XML form, it can be extracted from different databases and compared, so long as the two databases have equivalent fields, such as price and quantity. But what if the databases have fields that are similar but not equivalent? It would be a problem today, but perhaps not tomorrow. Emerging Web service innovations would add extra data, called metadata, that would let a database "announce" its structure. Then two different databases with similar fields could be compared by a software program with no human intervention at all [29].

Since Web services are used to create interoperable Web applications, there must be some mechanism to move XML data across the Internet. The easiest way would be to take advantage of an already existing protocol, the obvious candidate being the Hypertext Transport Protocol—the ubiquitous "http" part of a Web address. But HTTP was designed to move HTML data. For an Internet connection to transport XML instead of HTML for a Web service, a new mechanism was needed to allow XML data to piggyback on HTTP messages, the means by which Web sites receive commands from the keyboards of surfers and transmit data back for display. That mechanism is a new standard, Simple Object Access Protocol; developed by independent programmers in conjunction with researchers at Microsoft Corp., in Redmond, Wash [29]. Together, XML and SOAP give Web service applications unparalleled interoperability.

Web sites have to be able to announce to the service that they contain data—such as clearinghouse information, commodities listings, or an airline schedule— that might be useful to it. So another specification was developed: Universal Discovery, Description, and Integration. Basically, UDDI lets Web services look for databases in the same way that Google lets humans look for Web pages. One way that's done is

through UDDI registries, a Yellow Pages–like directory in which companies list their businesses and the Web-related services they provide [29].

Web Services Description Language is a standard that allows a machine to figure out on its own just what is at a site once it is been identified. A program accessing a Web service retrieves a WSDL description from the service. The description itself is specially formatted XML data telling the prospective user the procedures it can call and a little bit about them.

Figure A-0-1 shows an overview of the Web Services architecture and its typical workflow. The four specifications presented above: XML, SOAP, UDDI and WSDL are put together in the Web Services architecture to provide a solution to the heterogeneous application integration.

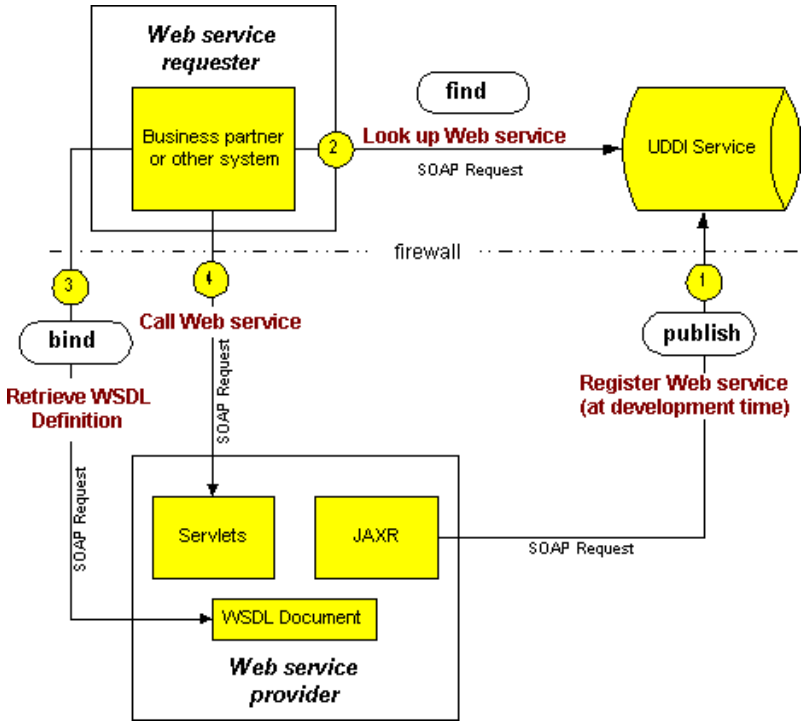


Figure A-0-1: Web services architecture

The Web service provider publishes its service using the UDDI service. Any Web service requestor, looking for a specific Web service, can query the UDDI service to find the desired service. As a result of this query, the requestor gets the location of the service endpoint where it can retrieve the WSDL service description. With the WSDL definition the Web service requestor can create a client application that can

interoperate with the Web service provider. The implementation of this client application has to be compliant with the WSDL description in order to call the Web service provider correctly.

Because of WSDL, a program calling a Web service can check the configuration of the Web service as the program runs, allowing the calling program to adjust for any changes that may have occurred in the Web service. This lets programmers separately develop and test the different components of an application, which will continue to run correctly even if one of its constituent modules is upgraded. Of course, benefits like these come at a price. There are extra run-time checks, and the text-based data used in XML makes it inefficient. So applications using Web services are several times slower than applications using binary data. In addition, sending plain-text XML across the open Internet makes it vulnerable to security breaches [29].

Table 2 shows the layered model for the Web Services architecture. The table contains the four fundamental specifications used in nowadays Web services implementations.

Table 2: The Web Services layered model.

Function	**Traditional Web Activities**	**Web Services**
Find a Web site	Search engine	UDDI
Site description	Search engine site description	WSDL
Transport protocol	HTTP	SOAP
Data format	HTML	XML

Appendix B: Configuration files

In this appendix, the most important configuration files can be found. These files are important to configure the programs used in the demonstration. Other files, like the WSDL definition of the TAPAS service, are presented for a better understanding of the communication process.

The first configuration file presented is the file used for the Apache Ant tool. This is a build file in XML used to make the compilation and deploying tasks easy. The Apache Ant tool is very useful in the development process of the implementation presented in this thesis.

```
<!--
    General purpose build script for web applications and web services,
    including enhanced support for deploying directly to a Tomcat 4
    based server.
    This build script assumes that the source code of your web application
    is organized into the following subdirectories underneath the source
    code directory from which you execute the build script:
      docs             Static documentation files to be copied to
                       the "docs" subdirectory of your distribution.
      src               Java source code (and associated resource files)
                       to be compiled to the "WEB-INF/classes"
                       subdirectory of your web applicaiton.
      web              Static HTML, JSP, and other content (such as
                       image files), including the WEB-INF subdirectory
                       and its configuration file contents.
    $Id: build.xml.txt,v 1.7 2002/12/28 09:08:58 jfclere Exp $
-->
<!-- A "project" describes a set of targets that may be requested
    when Ant is executed.  The "default" attribute defines the
    target which is executed if no specific target is requested,
    and the "basedir" attribute defines the current working directory
    from which Ant executes the requested task.  This is normally
    set to the current working directory.
-->
<project name="Plug-and-Play ws" default="compile" basedir=".">
<!-- ====================== Property Definitions =========================== -->
<!--
  Each of the following properties are used in the build script.
  Values for these properties are set by the first place they are
  defined, from the following list:
  * Definitions on the "ant" command line (ant -Dfoo=bar compile).
  * Definitions from a "build.properties" file in the top level
    source directory of this application.
  * Definitions from a "build.properties" file in the developer's
    home directory.
  * Default definitions in this build.xml file.
```

```
You will note below that property values can be composed based on the
contents of previously defined properties.  This is a powerful technique
  that helps you minimize the number of changes required when your development
  environment is modified.  Note that property composition is allowed within
"build.properties" files as well as in the "build.xml" script.
-->
  <property file="build.properties"/>
  <property file="${user.home}/build.properties"/>
<!-- ====================== File and Directory Names ========================== -->
<!--
  These properties generally define file and directory names (or paths) that
  affect where the build process stores its outputs.
  app.name             Base name of this application, used to
                       construct filenames and directories.
                       Defaults to "myapp".
  app.path             Context path to which this application should be
                       deployed (defaults to "/" plus the value of the
                       "app.name" property).
  app.version          Version number of this iteration of the application.
  build.home           The directory into which the "prepare" and
                       "compile" targets will generate their output.
                       Defaults to "build".
  catalina.home        The directory in which you have installed
                       a binary distribution of Tomcat 4.  This will
                       be used by the "deploy" target.
  dist.home            The name of the base directory in which
                       distribution files are created.
                       Defaults to "dist".
  manager.password     The login password of a user that is assigned the
                       "manager" role (so that he or she can execute
                       commands via the "/manager" web application)
  manager.url          The URL of the "/manager" web application on the
                       Tomcat installation to which we will deploy web
                       applications and web services.
  manager.username     The login username of a user that is assigned the
                       "manager" role (so that he or she can execute
                       commands via the "/manager" web application)
-->
  <property name="app.name"      value="myapp"/>
  <property name="app.path"      value="/${app.name}"/>
  <property name="app.version"   value="0.1-dev"/>
  <property name="build.home"    value="${basedir}/build"/>
  <property name="catalina.home" value="../../../.."/> <!-- UPDATE THIS! -->
  <property name="dist.home"     value="${basedir}/dist"/>
  <property name="docs.home"     value="${basedir}/docs"/>
  <property name="manager.url"   value="http://localhost:8080/manager"/>
  <property name="src.home"      value="${basedir}/src"/>
  <property name="web.home"      value="${basedir}/web"/>

  <property name="axis.app"      value="c:/Archivos de programa/jwsdp-1.3/webapps/axis/WEB-
INF/classes"/>
  <property name="fetched.dir"   value="${build.home}/fetched"/>
  <property name="generated.dir" value="${build.home}/generated"/>
<!-- ================== Custom Ant Task Definitions ========================== -->
<!--
  These properties define custom tasks for the Ant build tool that interact
  with the "/manager" web application installed with Tomcat 4.  Before they
  can be successfully utilized, you must perform the following steps:
  - Copy the file "server/lib/catalina-ant.jar" from your Tomcat 4
    installation into the "lib" directory of your Ant installation.
```

```
  - Create a "build.properties" file in your application's top-level
    source directory (or your user login home directory) that defines
    appropriate values for the "manager.password", "manager.url", and
    "manager.username" properties described above.
  For more information about the Manager web application, and the functionality
  of these tasks, see <http://localhost:8080/tomcat-docs/manager-howto.html>.
-->
  <taskdef name="install" classname="org.apache.catalina.ant.InstallTask"/>
  <taskdef name="list"    classname="org.apache.catalina.ant.ListTask"/>
  <taskdef name="reload"  classname="org.apache.catalina.ant.ReloadTask"/>
  <taskdef name="remove"  classname="org.apache.catalina.ant.RemoveTask"/>
<!--  ==================== Compilation Control Options ==================== -->
<!--
  These properties control option settings on the Javac compiler when it
  is invoked using the <javac> task.
  compile.debug        Should compilation include the debug option?
  compile.deprecation  Should compilation include the deprecation option?
  compile.optimize     Should compilation include the optimize option?
-->
  <property name="compile.debug"       value="true"/>
  <property name="compile.deprecation" value="false"/>
  <property name="compile.optimize"    value="true"/>
<!-- ==================== External Dependencies ============================ -->
<!--
  Use property values to define the locations of external JAR files on which
  your application will depend.  In general, these values will be used for
  two purposes:
  * Inclusion on the classpath that is passed to the Javac compiler
  * Being copied into the "/WEB-INF/lib" directory during execution
    of the "deploy" target.
  Because we will automatically include all of the Java classes that Tomcat 4
  exposes to web applications, we will not need to explicitly list any of those
  dependencies.  You only need to worry about external dependencies for JAR
  files that you are going to include inside your "/WEB-INF/lib" directory.
-->
<!-- Dummy external dependency -->
<!--
  <property name="foo.jar"
          value="/path/to/foo.jar"/>
-->
<!-- ==================== Compilation Classpath ============================ -->
<!--
  Rather than relying on the CLASSPATH environment variable, Ant includes
  features that makes it easy to dynamically construct the classpath you
  need for each compilation.  The example below constructs the compile
  classpath to include the servlet.jar file, as well as the other components
  that Tomcat makes available to web applications automatically, plus anything
  that you explicitly added.
-->
  <path id="compile.classpath">
    <!-- Include all JAR files that will be included in /WEB-INF/lib -->
    <!-- *** CUSTOMIZE HERE AS REQUIRED BY YOUR APPLICATION *** -->
<!--
    <pathelement location="${foo.jar}"/>
-->
    <pathelement location="${jaxp-api.jar}"/>
    <pathelement location="${dom.jar}"/>
    <pathelement location="${sax.jar}"/>
    <pathelement location="${xalan.jar}"/>
    <pathelement location="${jaxrpc-api.jar}"/>
```

```
    <pathelement location="${jaxrpc-spi.jar}"/>
    <pathelement location="${jaxrpc-impl.jar}"/>
    <pathelement location="${saaj-api.jar}"/>
    <pathelement location="${saaj-impl.jar}"/>
    <pathelement location="${ant.jar}"/>
    <pathelement location="${jaxb-api.jar}"/>
    <pathelement location="${jaxb-libs.jar}"/>
    <pathelement location="${jaxb-impl.jar}"/>
    <pathelement location="${jaxb-xjc.jar}"/>
    <pathelement location="${jaxr-api.jar}"/>
    <pathelement location="${jaxr-impl.jar}"/>
    <!-- Include all elements that Tomcat exposes to applications -->
    <pathelement location="${catalina.home}/common/classes"/>
    <fileset dir="${catalina.home}/common/endorsed">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${catalina.home}/common/lib">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${catalina.home}/shared/classes"/>
    <fileset dir="${catalina.home}/shared/lib">
      <include name="*.jar"/>
    </fileset>
  </path>
<!-- ==================== Clean Target ==================================== -->
<!--
  The "clean" target deletes any previous "build" and "dist" directory,
  so that you can be ensured the application can be built from scratch.
-->
  <target name="clean"
   description="Delete old build and dist directories">
    <delete dir="${build.home}"/>
    <delete dir="${dist.home}"/>
    <delete dir="${axis.app}/WsPlug-and-Play"/>
  </target>
<!-- ==================== Compile Target ================================== -->
<!--
  The "compile" target transforms source files (from your "src" directory)
  into object files in the appropriate location in the build directory.
  This example assumes that you will be including your classes in an
  unpacked directory hierarchy under "/WEB-INF/classes".
-->
  <target name="compile" depends="prepare"
   description="Compile Java sources">
    <!-- Compile Java classes as necessary -->
    <mkdir    dir="${build.home}/WEB-INF/classes"/>
    <javac srcdir="${src.home}"
          destdir="${build.home}/WEB-INF/classes"
            debug="${compile.debug}"
      deprecation="${compile.deprecation}"
         optimize="${compile.optimize}">
        <classpath refid="compile.classpath"/>
    </javac>

    <!-- Copy application resources -->
    <copy  todir="${build.home}/WEB-INF/classes">
      <fileset dir="${src.home}" excludes="**/*.java"/>
    </copy>
    <!-- Copy to the AXIS server -->
    <copy todir="${axis.app}">
```

```
    <fileset dir="${build.home}/WEB-INF/classes"/>
   </copy>
  </target>
<!-- ==================== Install Target ================================== -->
<!--
  The "install" target tells the specified Tomcat 4 installation to dynamically
  install this web application and make it available for execution.  It does
  *not* cause the existence of this web application to be remembered across
  Tomcat restarts; if you restart the server, you will need to re-install all
  this web application.
  If you have already installed this application, and simply want Tomcat to
  recognize that you have updated Java classes (or the web.xml file), use the
  "reload" target instead.
  NOTE:  This target will only succeed if it is run from the same server that
  Tomcat is running on.
  NOTE:  This is the logical opposite of the "remove" target.
-->
  <target name="install" depends="compile"
   description="Install application to servlet container">

    <install url="${manager.url}"
       username="${manager.username}"
       password="${manager.password}"
          path="${app.path}"
           war="file://${build.home}"/>
  </target>
<!-- ==================== Prepare Target ================================== -->
<!--
  The "prepare" target is used to create the "build" destination directory,
  and copy the static contents of your web application to it.  If you need
  to copy static files from external dependencies, you can customize the
  contents of this task.
  Normally, this task is executed indirectly when needed.
-->
  <target name="prepare">
    <!-- Create build directories as needed -->
    <mkdir  dir="${build.home}"/>
    <mkdir  dir="${build.home}/WEB-INF"/>
    <mkdir  dir="${build.home}/WEB-INF/classes"/>
    <mkdir  dir="${fetched.dir}"/>
    <mkdir  dir="${generated.dir}"/>

    <!-- Copy static content of this web application -->
    <copy todir="${build.home}">
     <fileset dir="${web.home}"/>
    </copy>

    <!-- Copy external dependencies as required -->
    <!-- *** CUSTOMIZE HERE AS REQUIRED BY YOUR APPLICATION *** -->
    <mkdir  dir="${build.home}/WEB-INF/lib"/>
  </target>
<!-- ==================== Import WSDL Target ============================== -->
<!--
   This target creates the proxy classes for the WS
-->
    <target name="import-wsdl-server">
    <java
     classname="org.apache.axis.wsdl.WSDL2Java"
            fork="true"
            failonerror="true"
```

```
            classpath="compile.classpath">
            <arg value="--server-side"/>
            <arg value="--skeletonDeploy"/>
            <arg value="true"/>
            <arg file="${fetched.dir}/wsPNES.wsdl"/>
     <arg value="--output"/>
     <arg file="${generated.dir}"/>
            <arg value="--verbose"/>
            <arg value="--package"/>
            <arg value="soaPlug-and-Playi"/>
    </java>
   </target>
<!-- ==================== Deploy Target ========================== -->
<!--
   This target deploys the service specified in the wsdd file into axis server
-->
  <target name="deploy">
    <java
      classname="org.apache.axis.client.AdminClient"
            fork="true"
            failonerror="true"
            classpath="compile.classpath">
            <arg value="-p"/>
            <arg value="80"/>
            <arg value="${build.home}/WEB-INF/deploy.wsdd"/>
    </java>
  </target>
<!-- ==================== Undeploy Target ========================== -->
<target name="undeploy">
    <java
      classname="org.apache.axis.client.AdminClient"
            fork="true"
            failonerror="true"
            classpath="compile.classpath">
            <arg value="${build.home}/WEB-INF/undeploy.wsdd"/>
    </java>
  </target>
</project>
```

Figure B- 1: File build.xml

To deploy the TAPAS service in the Apache Axis it is necessary a Web Service Deployment Descriptor (WSDD). The deploy.wsdd file presented in Figure B- 2 is compliant with this WSDD format and it is used by the deploy task of the Ant tool to deploy the service in Axis.

```
<!-- Use this file to deploy some handlers/chains and services      -->
<!-- Two ways to do this:                                           -->
<!--   java org.apache.axis.client.AdminClient deploy.wsdd          -->
<!--      after the axis server is running                          -->
<!-- or                                                             -->
<!--   java org.apache.axis.utils.Admin client|server deploy.wsdd   -->
<!--      from the same directory that the Axis engine runs         -->
```

```
<deployment
   xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

 <!-- Services from PNESInterfaceService WSDL service -->

 <service name="wsPNES" provider="java:RPC" style="rpc" use="encoded">
    <parameter name="wsdlTargetNamespace" value="http://localhost/axis/services/wsPNES"/>
    <parameter name="wsdlServiceElement" value="PNESInterfaceService"/>
    <parameter name="wsdlServicePort" value="wsPNES"/>
    <parameter name="className" value="soaPlug-and-Playi.WsPNESSoapBindingSkeleton"/>
    <parameter name="wsdlPortType" value="PNESInterface"/>
    <parameter name="allowedMethods" value="*"/>

    <typeMapping
      xmlns:ns="http://WsPlug-and-Play"
      qname="ns:Play"
      type="java:WsPlug-and-Play.Play"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
    <typeMapping
      xmlns:ns="http://xml.apache.org/xml-soap"
      qname="ns:mapItem"
      type="java:soaPlug-and-Playi.MapItem"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
    <typeMapping
      xmlns:ns="http://WsPlug-and-Play"
      qname="ns:RoleSession"
      type="java:WsPlug-and-Play.RoleSession"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
    <typeMapping
      xmlns:ns="http://WsPlug-and-Play"
      qname="ns:RequestResult"
      type="java:WsPlug-and-Play.RequestResult"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
    <typeMapping
      xmlns:ns="http://WsPlug-and-Play"
      qname="ns:GAI"
      type="java:WsPlug-and-Play.GAI"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
    <typeMapping
      xmlns:ns="http://WsPlug-and-Play"
      qname="ns:Role"
      type="java:WsPlug-and-Play.Role"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
```

```
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://WsPlug-and-Play"
  qname="ns:SubscribeRequest"
  type="java:WsPlug-and-Play.SubscribeRequest"
  serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
  deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://WsPlug-and-Play"
  qname="ns:RequestPars"
  type="java:WsPlug-and-Play.RequestPars"
  serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
  deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://WsPlug-and-Play"
  qname="ns:ApplicationMessage"
  type="java:WsPlug-and-Play.ApplicationMessage"
  serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
  deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://xml.apache.org/xml-soap"
  qname="ns:ArrayOf_tns1_GAI"
  type="java:WsPlug-and-Play.GAI[]"
  serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
  deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://WsPlug-and-Play"
  qname="ns:CapabilitySet"
  type="java:WsPlug-and-Play.CapabilitySet"
  serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
  deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://xml.apache.org/xml-soap"
  qname="ns:ArrayOf_xsd_int"
  type="java:int[]"
  serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
  deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://xml.apache.org/xml-soap"
  qname="ns:ArrayOf_xsd_string"
  type="java:java.lang.String[]"
  serializer="org.apache.axis.encoding.ser.ArraySerializerFactory"
  deserializer="org.apache.axis.encoding.ser.ArrayDeserializerFactory"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 />
 <typeMapping
  xmlns:ns="http://WsPlug-and-Play"
```

```
qname="ns:ActorPlugInReq"
      type="java:WsPlug-and-Play.ActorPlugInReq"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    />
  </service>
</deployment>
```

Figure B- 2: Web Service Deployment Descriptor deploy.wsdd

The TAPAS service definition in WSDL format is presented in Figure. Following this service definition, any developer under any platform can create its own client application to interact wit the service provider. The parameters used in the request and the logic of the remote procedure call are described in this file.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions targetNamespace="http://localhost/axis/services/wsPNES"
          xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:apachesoap="http://xml.apache.org/xml-soap"
          xmlns:impl="http://localhost/axis/services/wsPNES"
          xmlns:intf="http://localhost/axis/services/wsPNES"
          xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:tns1="http://Plug-and-Play"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:types>

  <schema targetNamespace="http://xml.apache.org/xml-soap"
          xmlns="http://www.w3.org/2001/XMLSchema">
          <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>

          <complexType name="Vector">
            <sequence>
                  <element maxOccurs="unbounded" minOccurs="0" name="item"
type="xsd:string"/>
            </sequence>
          </complexType>

          <complexType name="mapItem">
            <sequence>
                  <element name="key" nillable="true" type="xsd:string"/>
                  <element name="value" nillable="true" type="xsd:string"/>
            </sequence>
          </complexType>
```

```
        <complexType name="Map">
         <sequence>
                <element maxOccurs="unbounded" minOccurs="0" name="item"
type="apachesoap:mapItem"/>
         </sequence>
        </complexType>
        <complexType name="ArrayOf_xsd_string">
         <complexContent>
          <restriction base="soapenc:Array">
                <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]"/>
          </restriction>
         </complexContent>
        </complexType>

    <complexType name="ArrayOf_xsd_int">
         <complexContent>
          <restriction base="soapenc:Array">
                <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]"/>
          </restriction>
         </complexContent>
        </complexType>

        <complexType name="ArrayOf_tns1_GAI">
         <complexContent>
          <restriction base="soapenc:Array">
                <attribute ref="soapenc:arrayType" wsdl:arrayType="tns1:GAI[]"/>
          </restriction>
         </complexContent>
        </complexType>

  </schema>

  <schema targetNamespace="http://Plug-and-Play"
        xmlns="http://www.w3.org/2001/XMLSchema">
        <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>


        <complexType name="ActorPlugInReq">
         <sequence>
         <element name="location" nillable="true" type="tns1:GAI"/>
         <element name="role" nillable="true" type="tns1:Role"/>
         <element name="play" nillable="true" type="tns1:Play"/>
         <element name="rqCaps" nillable="true" type="tns1:CapabilitySet"/>
         <element name="rsCaps" nillable="true" type="tns1:CapabilitySet"/>
         </sequence>
        </complexType>

        <complexType name="GAI">
         <sequence>
                <element name="type" type="xsd:string"/>
                <element name="node" type="xsd:string"/>
                <element name="address" type="xsd:string"/>
                <element name="pas" type="xsd:string"/>
                <element name="name" type="xsd:string"/>
                <element name="hmhandles" type="apachesoap:Map"/>
                <element name="initialized" type="xsd:boolean"/>
         </sequence>
        </complexType>
```

```
        <complexType name="CapabilitySet">
          <sequence>
                <element name="capabilities" type="apachesoap:Vector"/>
                <element name="Add" type="xsd:int"/>
                <element name="Set" type="xsd:int"/>
                <element name="Remove" type="xsd:int"/>
          </sequence>
        </complexType>

        <complexType name="Play">
          <sequence>
                <element name="playId" type="xsd:string"/>
                <element name="playVer" type="xsd:string"/>
                <element name="playLoc" type="xsd:string"/>
                <element name="verA" type="xsd:int"/>
                <element name="verB" type="xsd:int"/>
          </sequence>
        </complexType>

        <complexType name="SubscribeRequest">
          <sequence>
                <element name="requestor" nillable="true" type="GAI"/>
                <element name="eventTypes" type="apachesoap:ArrayOf_xsd_int"/>
                <element name="PlayPlugIn" type="xsd:int"/>
                <element name="PlayChangesPlugIn" type="xsd:int"/>
                <element name="PlayPlugOut" type="xsd:int"/>
                <element name="ActorPlugIn" type="xsd:int"/>
                <element name="ActorPlugOut" type="xsd:int"/>
                <element name="ActorBehaviourPlugIn" type="xsd:int"/>
                <element name="ActorBehaviourPlugOut" type="xsd:int"/>
                <element name="ActorChangeBehaviour" type="xsd:int"/>
                <element name="ActorPlay" type="xsd:int"/>
                <element name="SubscribeRequest" type="xsd:int"/>
                <element name="SubscribeReport" type="xsd:int"/>
                <element name="SubscribeCancel" type="xsd:int"/>
                <element name="RoleSessionAction" type="xsd:int"/>
                <element name="ActorCapabilities" type="xsd:int"/>
                <element name="ActorCreate" type="xsd:int"/>
                <element name="ActorRemove" type="xsd:int"/>
                <element name="RoleSessionCreate" type="xsd:int"/>
                <element name="RoleSessionRemove" type="xsd:int"/>
                <element name="RT" type="apachesoap:ArrayOf_xsd_string"/>
                <element name="scope" type="apachesoap:ArrayOf_tns1_GAI"/>
                <element name="applType" type="apachesoap:ArrayOf_xsd_string"/>
                <element name="whenType" type="xsd:int"/>
                <element name="Immediately" type="xsd:int"/>
                <element name="Periodically" type="xsd:int"/>
                <element name="SpecifiedTime" type="xsd:int"/>
                <element name="WhenCancel" type="xsd:int"/>
                <element name="whenValue" type="xsd:int"/>
          </sequence>
    </complexType>

        <complexType name="ApplicationMessage">
      <sequence>
          <element name="roleSessionId" type="xsd:string"/>
          <element name="messageType" type="xsd:string"/>
          <element name="message" type="apachesoap:ArrayOf_xsd_string"/>
```

```
      </sequence>
    </complexType>

        <complexType name="RoleSession">
          <sequence>
                <element name="roleSessionId" nillable="true" type="tns1:RoleSession"/>
                <element name="initiator" nillable="true" type="tns1:GAI"/>
                <element name="cooperator" nillable="true" type="tns1:GAI"/>
          </sequence>
        </complexType>
     <complexType name="Role">
          <sequence>
                <element name="roleId" type="xsd:string"/>
          </sequence>
        </complexType>
        <complexType name="RequestResult">
          <sequence>
                <element name="FAIL" type="xsd:int"/>
                <element name="OK" type="xsd:int"/>
                <element name="removed" type="xsd:boolean"/>
                <element name="resCaps" nillable="true" type="tns1:CapabilitySet"/>
                <element name="resultCause" nillable="true" type="xsd:string"/>
                <element name="resultType" type="xsd:int"/>
                <element name="roleSession" nillable="true" type="tns1:RoleSession"/>
                <element name="subscribeIdentifier" nillable="true" type="xsd:string"/>
          </sequence>
        </complexType>
        <complexType name="RequestPars">
          <sequence>
                <element name="requestType" type="xsd:int"/>
                <element name="sender" nillable="true" type="tns1:GAI"/>
                <element name="receiver" nillable="true" tyep="tns1:GAI"/>
                <element name="PlayPlugIn" type="xsd:int"/>
                <element name="PlayChangesPlugIn" type="xsd:int"/>
                <element name="PlayPlugOut" type="xsd:int"/>
                <element name="ActorPlugIn" type="xsd:int"/>
                <element name="ActorPlugOut" type="xsd:int"/>
                <element name="ActorBehaviourPlugIn" type="xsd:int"/>
                <element name="ActorBehaviourPlugOut" type="xsd:int"/>
                <element name="ActorChangeBehaviour" type="xsd:int"/>
                <element name="ActorPlay" type="xsd:int"/>
                <element name="SubscribeRequest" type="xsd:int"/>
                <element name="SubscribeReport" type="xsd:int"/>
                <element name="SubscribeCancel" type="xsd:int"/>
                <element name="RoleSessionAction" type="xsd:int"/>
                <element name="ActorCapabilities" type="xsd:int"/>
                <element name="RT" type="tns1:ArrayOf_xsd_string"/>
                <element name="play" nillable="true" type="tns1:Play"/>
                <element name="actorPlugInReq" nillable="true" type="tns1:ActorPlugInReq"/>
                <element name="plugOutRoleSession" nillable="true"
type="tns1:RoleSession"/>
                <element name="plugOutActor" nillable="true" type="tns1:GAI"/>
                <element name="apo" type="xsd:boolean"/>
                <element name="upgradePars" type="tns1:ArrayOf_xsd_string"/>
                <element name="applicationMessage" nillabe="true"
type="tns1:ApplicationMessage"/>
                <element name="roleSession" nillable="true" type="tns1:RoleSession"/>
                <element name="subscribeRequest" nillable="true"
type="tns1:SubscribeRequest"/>
```

```
                    <element name="subscribeReport" type="apachesoap:ArrayOf_xsd_string"/>
                    <element name="subscribeCancel" type="xsd:string"/>
                    <element name="capOpType" type="xsd:int"/>
                    <element name="capabilities" nillable="true" type="tns1:CapabilitySet"/>
            </sequence>
           </complexType>
  </schema>
</wsdl:types>


  <wsdl:message name="syncRequestFromPNESResponse">
    <wsdl:part name="syncRequestFromPNESReturn" type="tns1:RequestResult"/>
  </wsdl:message>
  <wsdl:message name="syncRequestFromPNESRequest">
    <wsdl:part name="in0" type="tns1:RequestPars"/>
  </wsdl:message>
  <wsdl:portType name="PNESInterface">
    <wsdl:operation name="syncRequestFromPNES" parameterOrder="in0">
      <wsdl:input message="impl:syncRequestFromPNESRequest"
name="syncRequestFromPNESRequest"/>
      <wsdl:output message="impl:syncRequestFromPNESResponse"
name="syncRequestFromPNESResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="wsPNESSoapBinding" type="impl:PNESInterface">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="syncRequestFromPNES">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="syncRequestFromPNESRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://Plug-and-Play" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="syncRequestFromPNESResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://localhost/axis/services/wsPNES" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="PNESInterfaceService">
    <wsdl:port binding="impl:wsPNESSoapBinding" name="wsPNES">
      <wsdlsoap:address location="http://localhost/axis/services/wsPNES"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Figure B- 3: WSDL TAPAS service definition. File wsPNES.wsdl

The following files are used in the processes of publishing and discovery with the Registry Server. These files have to be present at each node in order to use the UDDI service adequately. The tapasconcepts.xml file, Figure B- 4, contains the uiid that identifies uniquely the TAPAS service at the Registry. And the tapasconcepts.dtd file,

Figure B- 1, specifies the format of the information provided at the publishing and discovery stages.

```
<?xml version="1.0" encoding="UTF-8"?>

<PredefinedConcepts>
<JAXRClassificationScheme id="uiid" name="TAPASScheme" >
<JAXRConcept id="uiid/TAPASServices" name="TAPASServices" parent="uiid" code="TAPASServices" />
<JAXRConcept id="uiid/wsPNES" name="TAPASServices" parent="uiid/TAPASServices" code="wsPNES" />
</JAXRClassificationScheme>
</PredefinedConcepts>
```

Figure B- 4: File tapasconcepts.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT PredefinedConcepts (JAXRClassificationScheme*)>
<!ELEMENT JAXRClassificationScheme (JAXRConcept*)>
<!ATTLIST JAXRClassificationScheme
                    id CDATA #REQUIRED
                    name CDATA #REQUIRED
                    description CDATA #IMPLIED
>
<!ELEMENT JAXRConcept (JAXRConcept*)>
<!ATTLIST JAXRConcept
                    id CDATA #REQUIRED
                    name CDATA #REQUIRED
                    parent CDATA #REQUIRED
                    code CDATA #IMPLIED
>
<!ELEMENT Command (JAXRClassificationScheme*, JAXRConcept*, namepattern*)>
<!ATTLIST Command
                    commandname CDATA #REQUIRED
                    path CDATA #IMPLIED
>
<!ELEMENT namepattern  (#PCDATA )>

<!ELEMENT Result (JAXRClassificationScheme*, JAXRConcept*)>
<!ATTLIST Result
                    commandname CDATA #REQUIRED
                    status CDATA #REQUIRED
                    error CDATA #IMPLIED
>
```

Figure B- 5: File tapasconcepts.dtd

The *Initial Service Request,* Figure B- 6, is the XML query that it is sent from the Director to the Selection Engine at the Service Plug-in process. This query specifies the Role figure to be plugged and the node that will instantiate it. The *Initial Service Request* is one of the request types considered at the Service Management framework, section 2.3. And it is the request used for the demonstration in chapter 5.

```
<InitialServiceRequest type="InitialServiceRequest">
        <sender />
        <dateTime />
        <serviceType>TeleSchool</serviceType>
        <roleRequesting>SchoolClient</roleRequesting>
        <preferredConfiguration>
                <nodeInstalling>http://comp1.tapas.org</nodeInstalling>
        </preferredConfiguration>
        <contextInfo>
                <connectionUsed>Bluetooth</connectionUsed>
                <userSubscription>Advanced</userSubscription>

                <MMSupport>Speaker</MMSupport>
        </contextInfo>
        <Result>
                <Manus>Svar_MName</Manus>
                <ActionGroup>Svar_Gi</ActionGroup>
                <Category>Svar_CapCategory</Category>
        </Result>
</InitialServiceRequest>
```

Figure B- 6: Initial Service Request.