Project Number:	**IST-1999-10561-FAIN**
Project Title:	**Future Active IP Networks**

D8: Final Specification of Case Study Systems

Editor:	**Alvin Tan / Marcin Solarski**
Document No:	**D8**
Contribution File Name:	**D8.doc**
Version:	**2.0**
Company:	**UCL/FHG**
Date:	**Tuesday, 13 May 2003**
Distribution:	**WP4**
Dissemination:	**CO**

The FAIN Consortium consists of:

Partner	Status	Country
UCL	Partner	United Kingdom
JSIS	Associate Partner to UCL	Slovenia
NTUA	Associate Partner to UCL	Greece
UPC	Associate Partner to UCL	Spain
DT	Partner	Germany
FT	Partner	France
KPN	Partner	Netherlands
HEL	Partner	United Kingdom
HIT	Partner	Japan
SAG	Partner	Germany
ETH	Partner	Switzerland
FHG/FOKUS	Partner	Germany
IKV	Associate Partner to GMD	Germany
INT	Associate Partner to GMD	Spain
UPEN	Partner	USA

Project Management

Alex Galis
University College London
Department of Electronic and Electrical Engineering,
Torrington Place
London WC1E 7JE
United Kingdom
Tel +44 (0) 207 458 5738
Fax +44 (0) 207 388 9325
E-mail: a.galis@ee.ucl.ac.uk

Authors

Alvin Tan (UCL) – Contributor/Editor (Management)
Marcin Solarski (FHG) – Contributor/Editor (ASP)
Celestin Brou (FHG) - Contributor
Epifanio Salamanca (UPC) - Contributor
Edgar Magaña (UPC) - Contributor
Julio Vivero (UPC) - Contributor
Juan Luis Mañas (INT) – Contributor
Christos Tsarouchis (HEL) – Contributor
Chiho Kitahara (HIT) - Contributor
Yiannis Nikolakis (NTUA) – Contributor
Eun-Mok Lee (FHG) – Contributor
Matthias Bossardt (ETH) – Contributor
Yannick Carlinet (FT) – Contributor
Bertrand Mathieu (FT) – Contributor
Richard Lewis (UCL) – Contributor
Alex Galis (UCL) - Contributor

Change History

Version	Data	Authors	Comments
0.0	17/03/03	Epi Salamanca (UPC)	Initial TOC of the Management Part for D8
0.1	30/04/03	Epi, Edgar Magaña (UPC) Christos Tsarouchis(Hit)	Overall re-editing and structure Delegation PDP and PEPs PBNM general description
0.2	30/04/03	Alvin Tan (UCL)	UCL Contribution Reformatting style
0.3	30/04/03	Epì	Contribution from UPC- Epi – API for NMS.
0.4	02/05/03	Alvin	IDM APIs, further formatting
0.5	06/05/03	Alvin, Epi, Edgar, Juan-Luis Manas (INT)	EMS API Section New Figures for EMS and NMS components Modified PDP Manager Section Enhancement of the Core and EMS Section.

			Monitoring component updated.
0.6	07/05/03	Christos, Chiho, Juan-Luis, Alvin	Del PDP updates, EMS/NMS monitoring enhancement Comments on material
1.0	08/05/03	Yiannis Nikolakis (NTUA) Julio Vivero (UPC) Epi, Juan-Luis Alvin	RM section inserted Informal review Policy parser, policy editor sections included First complete version contains all the material required for the deliverable.
1.1	08/05/03	Alvin Tan	ASP and MS merge
1.2	09/05/03	Marcin Solarski	ASP component structure update
1.3	09/05/03	Alvin Tan Edgar Magana	QoS PDP at NL updated. Further corrections
1.4	10/05/03	Marcin Solarski Eun-Mok Lee	Better structure for 3.5.1 (each component in separate section) Updated ASP architecture (sec 3.1) Updated Network ASP (sec 3.5.1) Updated Node ASP (sec 3.5.2)
1.5	10/05/03	Richard Lewis	Overall review
1.6	11/05/03	Celestin Brou	Introduction
1.7	12/05/03	Alex Galis	Overview Overall review
2.0	13/05/03	Epi Alvin / Marcin	Use cases at element and network level Final version

TABLE OF CONTENTS

TABLE OF MAIN ACRONYMS

ANSP	Active Network Service Provider
ASP	Active Service Provisioning
CM	Code Manager
CORBA	Common Object Request Broker Architecture
IDM	Inter-Domain Manager
MbD	Management by Delegation
NIP	Network Infrastructure Provider
PBNM	Policy-Based Network Management
PCIM	Policy Core Information Model
PDP	Policy Decision Point
PEP	Policy Enforcement Point
RCF	Resource Control Framework
RM	Resource Manager
SCE	Service Creation Engine
SLA	Service Level Agreement
SM	Service Manager
SP	Service Provider
VAN	Virtual Active Network
VE	Virtual Environment
VEM	Virtual Environment Manager
XML	Extensible Mark-up Language

1. OVERVIEW AND INTRODUCTION

In this deliverable, we described a hierarchically distributed policy-based network management architecture (Chapter 2) and an Active Service Provisioning architecture (Chapter 3), which are important results of the FAIN project. Conclusions and further work is presented in Chapter 4. These architectural components are depicted Figure 1.

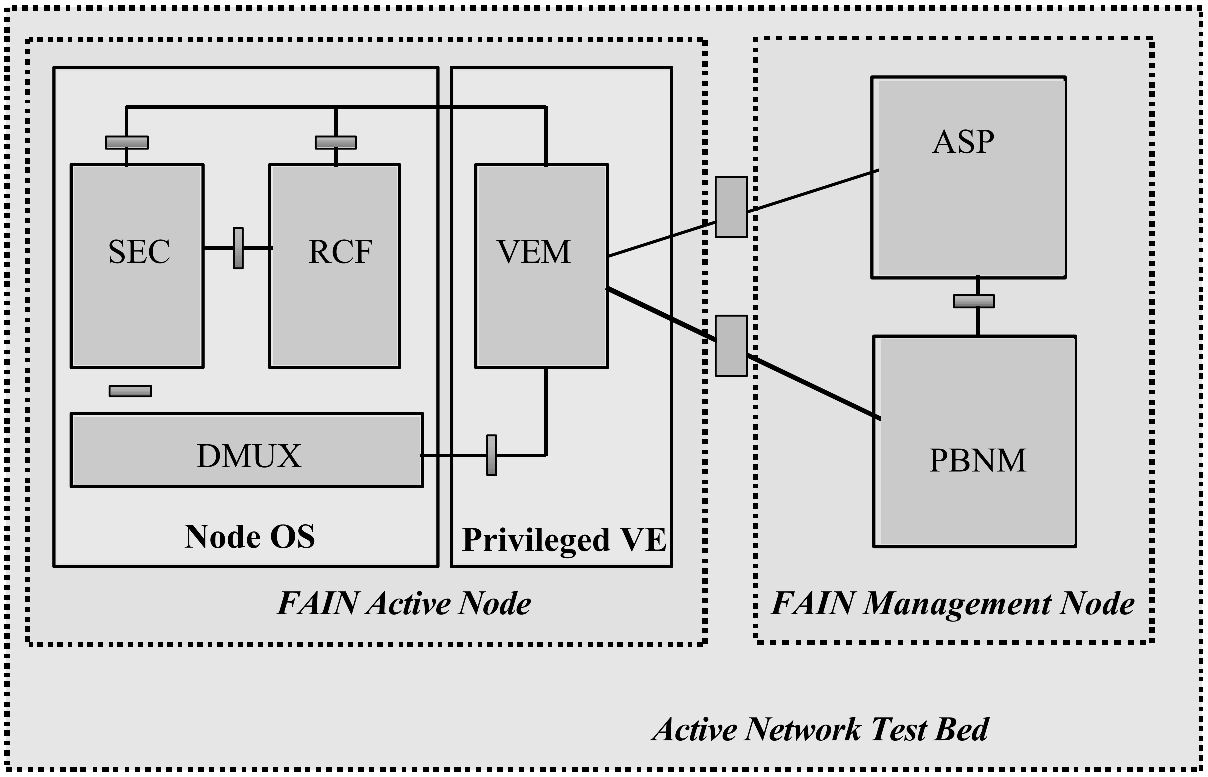


Figure 1. Overview of FAIN Active Nodes and Management Nodes

Each of these architectural components is provided by one of the two sub-systems that constitute the FAIN Management System, briefly described below.

The **policy-based management system**, with appropriate policies, performs fine grain management of the FAIN active node resources and delegates management capability to third-parties, according to the Fain Business Model service chain. Thus the Active Network Service Provider (ANSP) delegates management functionality to its registered Service Providers (SP) that in turn delegates particular management tasks to their customers. Security and isolation of resource usage are assured by mechanisms developed by the FAIN Project and deployed in the FAIN Active Node. The Uniform API offered by The FAIN Node allows the deployment of a manufacturer independent network-wide solution.

The **ASP** system provides the mechanisms necessary to organise registered service components in accordance with their run-time environment constraints, composes them on-demand into a given service, and deploys the service over the FAIN network. Thus ASP constitutes an important FAIN architectural component, which demonstrates activeness by deploying components that program network nodes as a way of provisioning new services for a given purpose.

Together the ASP and PBNM provide all the facilities required to create a tailored Virtual Access Network (VAN), operating over the FAIN Active Network. Realisation of these systems is based on the architecture developed earlier in the project, which encompasses the local node (network element), the individual management domains in isolation and, latterly, the inter-domain management issues. Within the architecture, the complexity of the system being developed is managed by decomposing the system into sub-systems that collaborate to achieve the global FAIN management goals: the Active Node level sub-systems (node-ASP and EMS), and the network level sub-systems (Network Level Management System and Network level ASP), all of which are further decomposed.

Further analysis of the NMS identified generic policy-based management functionality at Element (node) level and Network level to be specialised into both levels with their semantically particularities. These generic functionalities are designed and developed in a component named "CORE sub-system" in the Policy-Based Network Management chapter. The figure below represents this view of FAIN Management System Architecture.

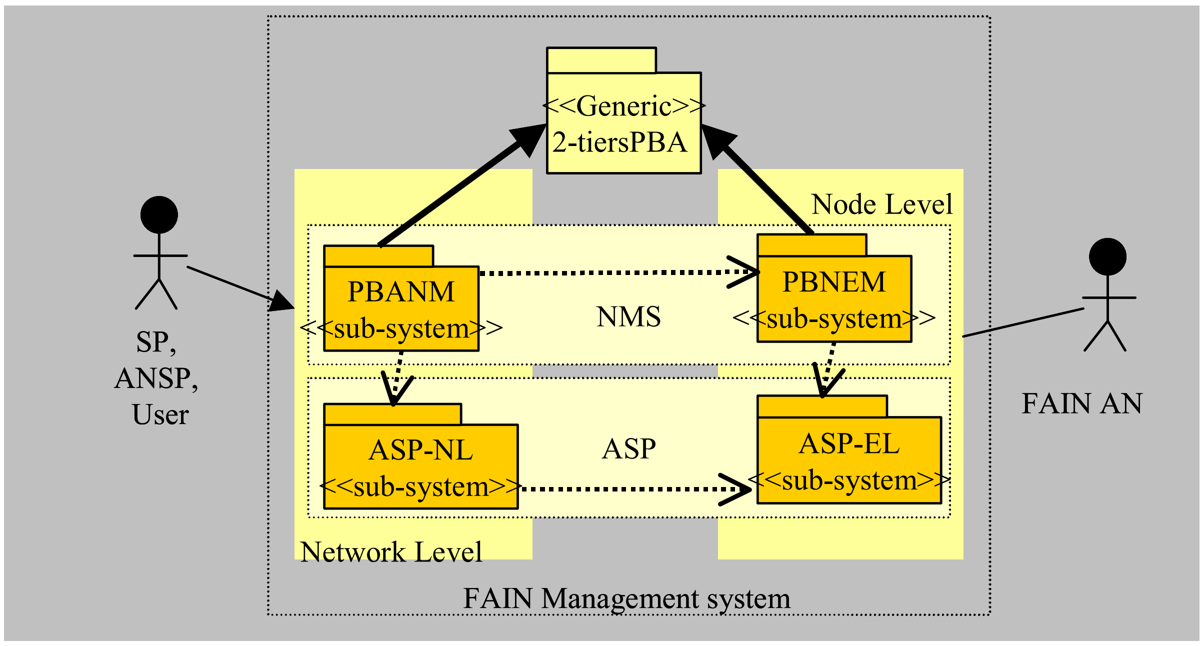


Figure 2. Use case view of the management system in FAIN

We have applied policies as a way of managing active networks and we have used active technologies and mechanisms to extend the management architecture by dynamically deploying additional PDPs and PEPs.

Although PDPs and PEPs can be deployed on demand, they must comply with the expected (standardised) interface and be registered in the ASP system. Also, our management architecture supports an extension mechanism of a finer granularity by dynamically adding new functionality (policy action and condition interpreters) into already existing PDPs/PEPs.

We have used different types of PDP and PEP as a means of differentiating groups of policies and facilitating policy decision-making according to a specific context.

Based on a new Business Model that advocates the deployment of virtual networks on top of the same network infrastructure, we have extended the concept of management by delegation by allowing multiple management architectures to be instantiated and to function independently of each other. This was enabled by the use of the FAIN active node and its open interface.

Finally, we have mostly focused on implementing and experimenting with the configuration model for policy control. We consider the outsourcing model to be equally important. According to this model, control protocols must be policy-aware in order to convey policy information that is necessary for the PDPs to make a decision. In addition, the PDPs need to interact with the PEPs, therefore additional semantics must be built into the protocol to enable communication with a particular PEP. Building protocols with these properties is also one of the aims of our next stage of research.

1.1. FAIN PBNM Management Architecture

The FAIN PBNM management architecture is designed as a hierarchically distributed architecture, consisting of two levels (two-tiered architecture): the network management level, which encompasses the Network Management System (NMS) and the element management level, which encompasses the Element Management System (EMS).

Furthermore, the defined policies have been categorised according to the semantics of management operations, which may range from QoS operations to service-specific operations. Accordingly, policies that belong to a specific category are processed by dedicated Policy Decision Points (PDPs) and Policy Enforcement Points (PEPs).

The NMS is the entry point of the management architecture. It is the recipient of policies, which may have been the result of network operator management decisions or of service level agreements (SLA) between ANSP & SP, or SP & C. These SLAs require reconfiguration of the network, which is automated by means of policies sent to the NMS.

Network-level policies are processed by the NMS PDPs, which decide when policies can be enforced. When enforced, they are delivered to the NMS PEPs that map them to element level policies, which are, in turn, sent to the EMSs. EMS PDPs perform similar processes at the element level. Finally, the AN node PEPs execute the enforcement actions at the NE.

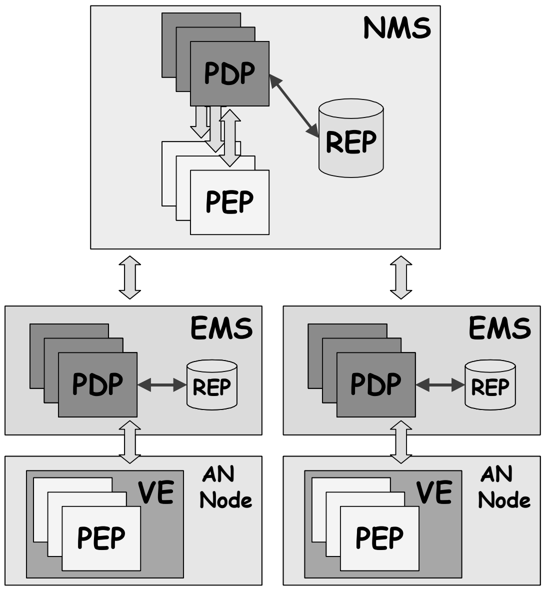


Figure 3. A hierarchical view of the FAIN Management Architecture

The use of this "policy control configuration model" [8] and its use in a hierarchically distributed management architecture combines the benefits of management automation with reduction of management traffic and distribution of tasks.

As the FAIN management architecture is based on the FAIN Business Model, the relationship among the three main actors, namely, ANSP, SP, and C, is projected directly onto the architecture. Accordingly, each one of these actors may request and get his own (virtual) management architecture through which he is enabled to manage the resources allocated to the Virtual Environments (VE) of his Virtual Network.

In this way, each actor is free to select and deploy his own model of managing the resources, namely his own management architecture, which can be centralized, hierarchical, policy or non-policy based. The complexity of the virtual network and the types of service that are deployed in it, dictate the particular choice of management architecture by its owner. In addition, different management architectures simultaneously coexist in the same physical network infrastructure as they may be deployed by different actors. To this end, we create an environment that is capable of accommodating opposing requirements, an accomplishment that is beyond the capabilities of the traditional approach of monolithic architectures.

Our model extends the Tempest approach [9] to the management plane, which was the first to advocate the simultaneous support of (virtual) control architectures for ATM networks.

It also extends the scope of Management by Delegation (MbD) [10] as it allows delegation of the network management responsibility to a third party, e.g. an SP, which can be deployed and hosted in a separate physical location from the NMS of the owner of the network, e.g. the ANSP.

Figure 4 illustrates the aforementioned discussion. Starting with the management architecture of the network operator, namely the ANSP, it instantiates and registers a new Management Instance (MI), which is delegated to one of his customers, i.e. the SP. This management instance will host the SP's management architecture. The SP has the option to buy from the ANSP an instance of the ANSP's architecture, in our case a policy-based one. To this end, the network management architecture developed by the ANSP is not only used for managing the Network Elements (NEs) but it becomes a commodity, thus creating another important source of income for the ANSP.

Furthermore, the ability of the ANSP to generate and support multiple management domains may create additional business opportunities. For example, the ANSP may build an OSS hosting facility for SPs to instantiate their own management architectures. In this way, the ANSP may sell both his expertise in running and operating an OSS as well as the architecture and its corresponding implementation.

In contrast, the SP does not need to build his management architecture from scratch but can customise an existing one according to the services he intends to run. Alternatively, the SP may deploy his own management architecture using the OSS hosting facility provided by the ANSP, so reducing the cost of managing the network.

In FAIN we have focused and experimented with the automated instantiation of management architectures using as a blueprint the PBNM system of the ANSP to instantiate another management system for the SP. Note also that this instantiation relationship can be recursive in the sense that the SP may further delegate his own instances to a Consumer.

Finally, the architecture of the MI used by the ANSP has been designed in such a way that it is dynamically extensible in terms of its functionality, as a result of using active networks technology.

The ANSP's management architecture can be extended in two distinct ways: a) deployment of a whole new pair of PDP/PEPs that implement new management functionality, or b) extension of the inner functionality of existing PDP/PEPs. The former is triggered by the PDP Manager whereas the latter is achieved by the PDPs themselves. The execution of the extension, namely fetching and deploying the requested functionality, is the responsibility of FAIN's Active Service Provisioning (ASP) system [1].

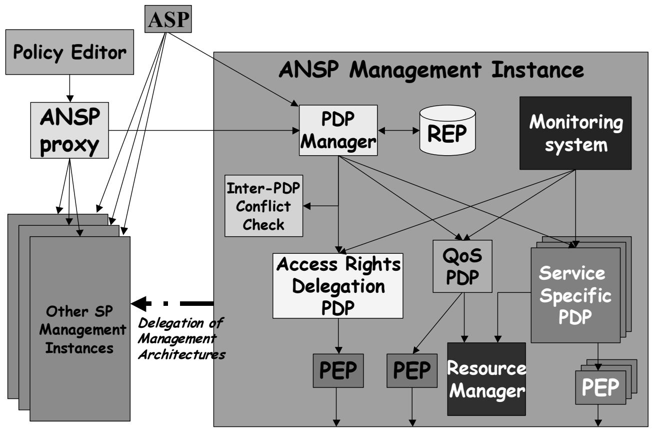


Figure 4. FAIN Management Instances and their Components

One important assumption underlying the previously described virtual management architectures is that well-established open interfaces and protocols have to be provided by the NEs. This may seem from the outset to be a demanding condition but there is convincing evidence that there exists a strong push towards ubiquitous open interfaces. Initiatives like the IEEE P1520 and lately the IETF ForCES working group serve as a proof for such claims. Furthermore, the programmable and active networks paradigm also relies on similar assumptions [12].

The FAIN prototype implementation is deployed on the pan-European FAIN testbed, an overlay network connecting ten different sites. Initial trials have focused mainly on functional evaluation of our management system, and in particular on the creation and usage of MIs and their extensibility features.

1.2. FAIN Active Service Provisioning Architecture

Active Service Provisioning (ASP) is understood in the context of the FAIN project as system aiming at deploying active services in the FAIN network. In general, active service deployment is considered as a process of making a service available in the active network so that the service user can use it. The deployment process is usually seen as a number of preparatory activities before the phase of the service operation. The typical activities include releasing the service code, distributing the service code to the target location, installing it and activating it.

Since the mid nineteen-nineties many efforts have been made to develop Active Networks technology to enable more flexibility in provisioning services in networks. By defining an open environment on network nodes this technology allows to rapidly deploy new services which otherwise may need a long time and adoption of hardware.

The FAIN project follows an approach in which a number of existing and emerging active network technologies are integrated. With regard to deployment, it proposes a novel approach to deploying services in heterogeneous active networks. In particular, the FAIN approach to deployment is characterised by the following:

- On-demand service deployment support. The ASP supports deploying a service whenever it is needed. A service deployment may be explicitly requested by a service provider or by another service already deployed or a management component.
- Component-based approach. Deploying and managing high-level services requires an appropriate service model. While fully-fledged component-based service models are an integral part of many enterprise computing architectures (e.g. Enterprise JAVA Beans, CORBA Component Model, Microsoft's.NET), it is not the case in many approaches developed by the active networking community. The FAIN deployment framework is designed on top of a component-based service model similar to the CORBA Component Model. The service model is hierarchical in that service components may recursively include sub-components. This allows for a fine-grained service description and composition.
- Network and node level architecture. To deal with complexity of deployment issues in active networks, the Active Service Provisioning has been designed according with rule of separation of concerns. The network-level ASP copes with network issues that include finding the nodes of the target environment for a given service considering topological service requirements as well as network link QoS requirements, for instance bandwidth. The node-level ASP, on the other hand, is concerned with node specific requirements, including technology and other service dependencies.
- Integrated Service Deployment and Management. The FAIN approach to service deployment is tightly integrated to FAIN service and network management. On one hand, the ASP depends on the service management framework implementing EE-specific deployment mechanisms, including installation and instantiation. On the other hand, the target environment in which the service is to be deployed are co-determined by the Network Management System The target environment is defined to be a Virtual Active Network which is established by the FAIN Network Management System. The VAN is created by the management system according to the service requirements.
- Selective code deployment. The service code distribution is done by selective downloading selected code modules from a code repository. The decision as to which code modules are needed is made at the ASP components at the target active nodes.
- Support for heterogeneous services and networks. The ASP has been designed to enable service deployment in heterogeneous networks. This is achieved by specifying an unified interface to the node capabilities and a unified notation for describing service specification and the implementation requirements. Whereas the CORBA technology is used to define the unified API to the node, the XML technology is used to define the unified service description.

The main actors communicating with the ASP system are:

- **Service Provider,** or SP for short, composes services that include active components and deploys these components in the network via the Active Service Provisioning, and offers the resulting service to Consumers. The service provider is responsible for releasing and withdrawing a service which includes a service version update or a complete remove of the service from specific nodes or from the complete active network respectively. Furthermore, the SP may be represented by the FAIN Network Management System with regard to initiation of service deployment or service reconfiguration.
- **Active Network Service Provider,** or ANSP for short, provides facilities for the deployment and operation of the active components into the network. Such facilities come in the form of an active middleware, support of new technologies. ANSP is represented by Active Nodes which are the target environment in context of deployment, which means that services may be deployed in these nodes and use the node resources made available to them by the ANSP.

These roles are described in the FAIN Enterprise Model in more detail in Deliverable D1. The main use cases of the ASP system are:

- Releasing a service. The Service Provider who decides to offer his service in the active network has to release it in the active network. The service is released by making the service meta-information and service code modules available to the ASP system.
- Deploying a service. After the service is released in the network, the Service Provider may want to deploy his service so that it can be used by a given service user. It means finding a target nodes that are most suitable for the given service installation, determining a mapping of the service components to the available Execution Environments of the target node, downloading the appropriate code modules, and finally installing and activating them.
- Reconfigure Service. The Service Provider or Network Management System on his behalf may request changing the current configuration of the service. It may include modifying component bindings, deploying additional service components or redeploying components that have been already deployed.
- Removing a service. The Service Provider may request to remove a deployed service from the environment it was deployed in. The ASP identifies the installed service components and removes them from the EEs of the target environment.
- Withdrawing a service. A service released in the active network may be withdrawn so that is no longer available to be deployed. The ASP removes the service meta-information and discards the service code modules.

The FAIN ASP system has a two-layered architecture: the network level and node level. The network level functionality is concerned with finding the target nodes for the service to deploy, coordination of the deployment process at the node level, and providing a service code retrieval infrastructure. At the node level, necessary service components are identified through code dependency resolution as well as the deployment mechanisms, including service installation, activation and pre-configuration are controlled.

The network level ASP system consists of three components: Network ASP manager, Service Registry and Service Repository.

The Network ASP Manager serves as an access component to the ASP system. In order to initiate the deployment of an particular service, a Service Provider contacts the Network ASP Manager and requests a service to be deployed as specified by the service descriptor.

The Service Registry is used to manage service descriptors. Service descriptors are stored on it, when a service component is released in the network. Network ASP Manager and the Service Creation Engine (described below) may contact the Service Registry to fetch service descriptors. Finally, service descriptors are deleted from the Service Registry, if a service is withdrawn from the network. In figure 3 only one Service Registry is shown in the network. Of course, several instances with possibly different content could be deployed in a network.

The Service Repository is a server for code modules. A code module is stored on the Service Repository, when a service descriptor referencing the particular code module is released in the network. The Code Manager, which is part of the node level ASP system and is described below, may fetch code modules from the Service Repository. A code module is deleted, if a service descriptor referencing the particular code module is withdrawn. As is the case for the Service Registry, several Service Repositories may coexist in a big network.

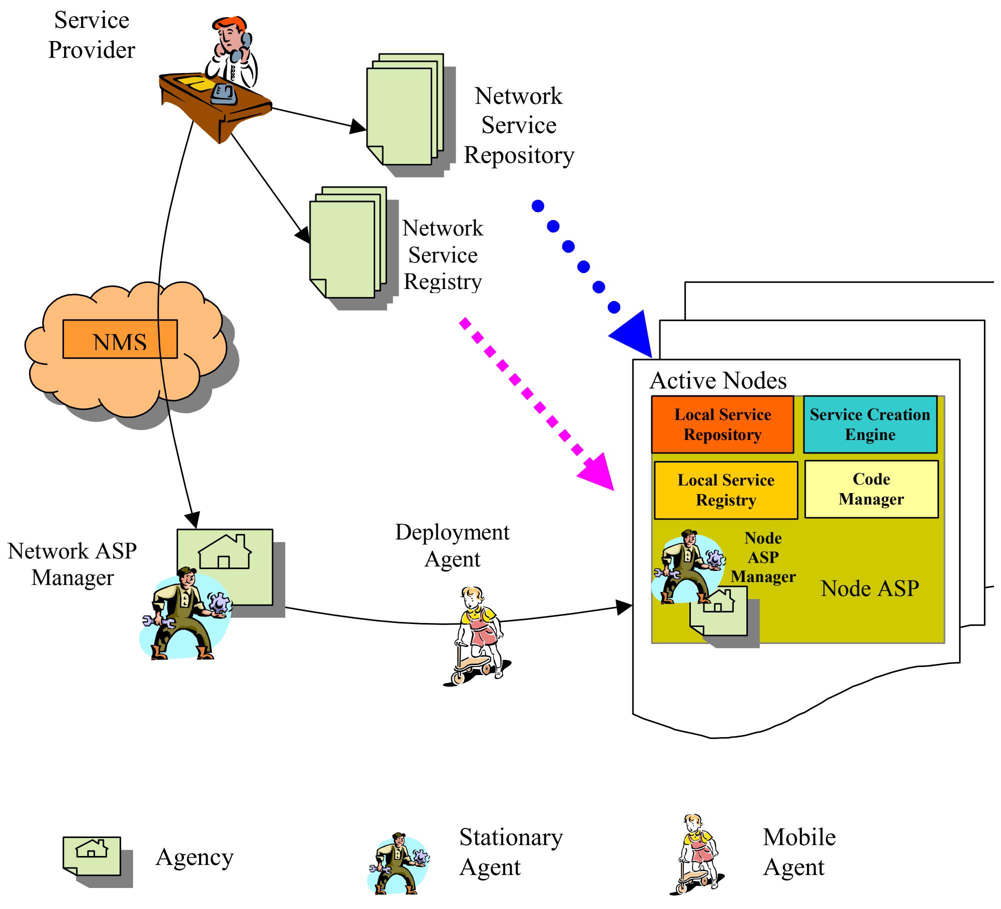


Figure 5. FAIN Active Service Provisioning

Node Level ASP Design. On the node level, the following components make up the ASP system as shown in the node ASP block: Node ASP manager, Service creation engine and Code Manager.

The Node ASP Manager is the peer component to the network ASP manager on the node level. The network ASP manager communicates with the node ASP manager in order to request the deployment, upgrading and removal of service components. The requests are dispatched to the service creation engine, or the code manager, respectively, which implement corresponding methods.

The Service Creation Engine plays a major role in the node level deployment of service components. Its main task is to select appropriate code modules to be installed on the node in order to perform the requested service functionality. The service creation engine matches service component requirements against node capabilities and performs the necessary dependency resolution. Since the service creation engine is implemented on each active node, active node manufacturers are enabled to optimise the mapping process for their particular node. In this way it is possible to exploit proprietary, advanced features of an active node.

The selection of service components is based on service descriptors that are retrieved from the service registry. As a result information about code modules that are to be installed on the particular node (a so-called service tree) is passed to the Code Manager.

The Code Manager performs the execution environment independent part of service component management. During the deployment phase, it fetches code modules identified by the service tree from the service repository. It also communicates with Node Management to perform EE-specific part of installation and instantiation of code modules. The Code Manager maintains a database containing information about installed code modules and their association with service components. If a particular service component needs to be removed this database is consulted in order to find out which code modules are associated with the component and, as a consequence, must be removed as well.

Please note that information fetched from service registry and repository is locally stored in their respective cache (local service registry, local service repository) in order to optimise recurrent service deployment requests.

2. POLICY-BASED NETWORK MANAGEMENT SYSTEM

2.1. Introduction

This chapter describes the network management system that was developed in FAIN. Together with the Active Service Provisioning (ASP) system, the FAIN network management system, namely the network-level management system and the element-level management system, constitutes the FAIN management system. Within the network and element management systems, we identify common 'core' components that represent the novelty of our policy-based approach. We first describe these core components before proceeding to explain how the network and element-level implementations inherit from the common features particularising them in order to cope with the expected functionalities at each level.

2.2. Common Components

The following section introduces, through a set of use cases, the basic functionalities of this core policy-based management sub-system. The next section describes all the components that form this core.

2.2.1 Common Use Cases

This section describes the main common use cases of the core management system, which are shown in the use case diagram in Figure 6.

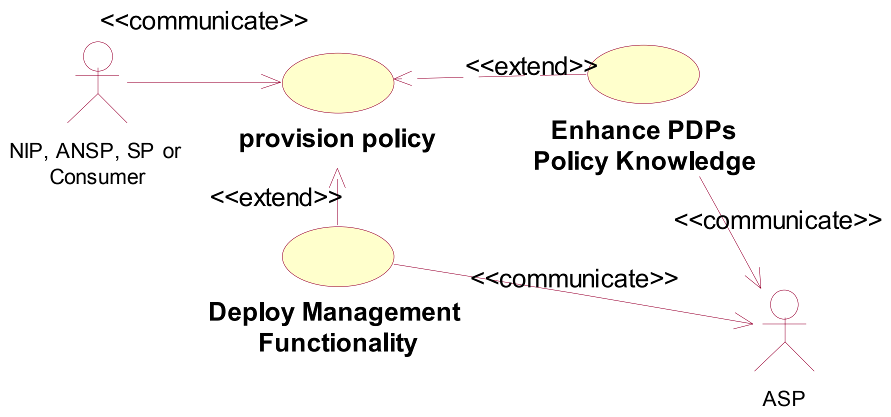

<<communicate>>
NIP, ANSP, SP or Consumer
provision policy
<<extend>>
Enhance PDPs Policy Knowledge
<<extend>>
<<communicate>>
Deploy Management Functionality
<<communicate>>
ASP

Figure 6. Use Case Diagram for Common Components

All the functionalities represented by these use cases are supported by the core policy-based management framework, and therefore by both the network-level and element-level management systems which extend from it.

2.2.1.1 *Provision Policy*

This is probably the most important use case for a policy-based management system. It represents the basic policy processing functionality. That is, the ‘provision policy’ use case encompasses all functionalities realised in our management framework each time a policy is introduced in the system. The activity diagram in Figure 7 shows the main functionality within the provisioning use case.

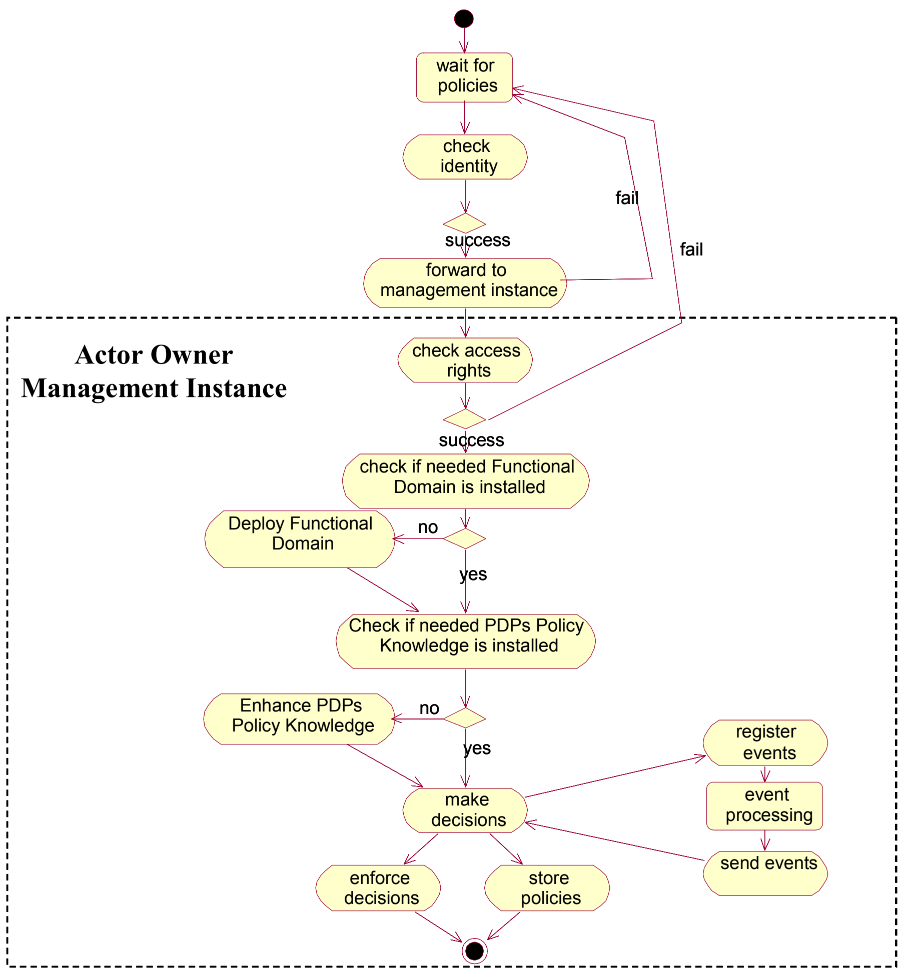


Figure 7. Activity Diagram for Core PDP Component

First, the pre-processing functionality, which is realised outside any management instance[1] checks the identity through its credentials, of the actor that intends to use the management system and demultiplexes the policy to the corresponding management instance.

[1] A management instance can be seen as a sandbox where all components running have the same owner. Each management instance has, at least, one component: the PDP Manager. For the ANSP it is formed by a PDP Manager, a QoS PDP/PEP and a Delegation of Access Rights PDP/PEP. All these components will be described in more detail later on.

Once the policy is dispatched to a particular management instance the steps that will be followed are as follows:

- Check the actor rights within the management instance. Each management instance has an associated profile. Here, we define what the actor is allowed to do and the maximum amount of resources that can be allocated. This profile has been implemented as an XML schema used to validate the incoming request in the form of XML policies.
- Extend the management functionality through the download of new components to correctly process the policy. This feature is explained by the use case called 'Deploy Management Functionality'.
- Where necessary, extend the management functionality of the PDP by upgrading the action and condition interpreters. The 'Enhance PDP's Policy Knowledge' use case further explains this functionality.
- Execute the core policy functionality (the list below identifies the most common functionality in a policy-based system):
    - check policy syntax and semantic conflicts
    - store policy in the repository
    - make decisions about when a policy should be enforced based on events received through the event processing functionality
    - enforce decisions

2.2.1.2 Deploy Management Functionality

Another basic feature of an Active Networks management system is its ability to extend itself with functionality, unforeseen at development time. For this reason, it requires an appropriate mechanism to add new functionality a run-time.

Once the management system detects that it must be extended, it requests the ASP framework to deploy the required functional domain[2]. Henceforth, the functionality to forward the request begins.

2.2.1.3 Enhance PDP's Policy Knowledge

This use case is an enhancement from the previous one. The management system is able to accept new policies from an already existing functional domain by triggering the deployment of new action/condition interpreters.

The Fain Policy Rules are PCIM[3]-compliant. The system extracts the appropriate fields to determine which class is responsible for interpreting the condition and action fields included in the incoming policy request. If the system detects that the required class this is not located in the local system it issues a request to the ASP to transport the corresponding code package from the network code repository to the local code repository. Henceforth the functionality for processing the request is resumed.

[2] Functional domain must be understood as all required components needed for processing policies that have conditions and actions that conceptually address a common management goal i.e. QoS, Delegation of Access Rights, and performance.

[3] The Policy Core Information Model (PCIM) describes the generic policy entities (policy groups, rules, conditions and actions) and their relationships in a domain-independent manner. Appropriate extensions are required in order to apply the PCIM to specific domains, such as QoS or security

2.2.2 Common Components Description

This section introduces the core components, depicted in Figure 8, that form part of the core policy-based management system. The core components are used at both management levels, together with level-specific components which might extend the core functionality and so realise level specific functionality.

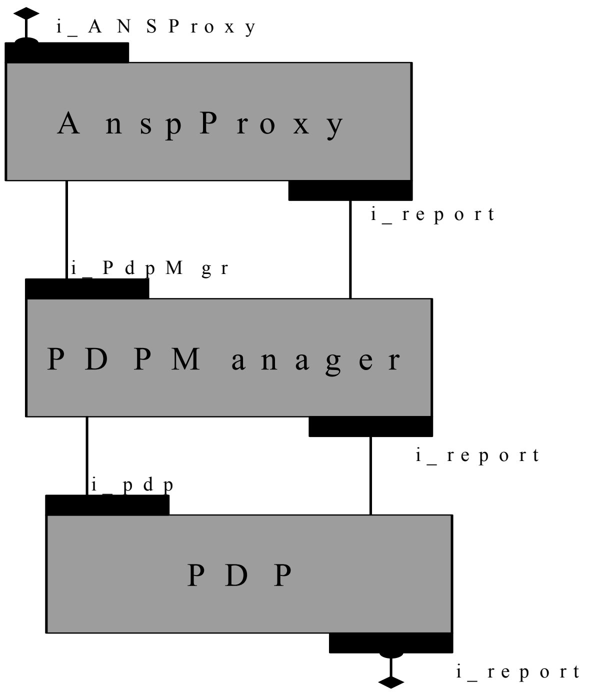


Figure 8. Architectural Model for Core System

In the following sub-section we will describe, these common components and their main capabilities.

2.2.2.1 ANSP Proxy

Policies originating from the policy editor are sent to the Network level ANSP proxy. The proxy has been introduced to enhance the security of the ANSP and/or of its customers, the SPs. It provides authentication of the incoming requests (policies) and forwards the policies to the correct management instances (MIs). Additionally, the element-level ANSP proxy receives policies from the network-level PEPs and forwards them to the PDP Manager at the element-level.

The ANSProxy can accept policies coming from both the ANSP and the SP, even directly from customers (end-users). The figures below show the case where policies are introduced into the ANSP proxy through the Policy editor, however the users may be allowed to use their own facilities to create policies and send them to the ANSP proxy directly. The NL PDP Manager always sends a report to the ANSP proxy, that contains the policy deployment status. The ANSProxy forwards this report to the Policy Editor. The use case diagram of the NL ANSPproxy is shown in Figure 9.

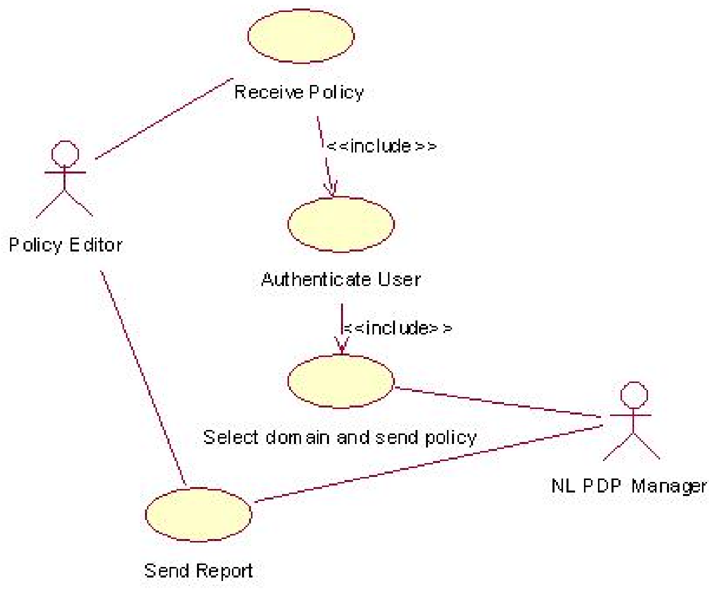


Figure 9. Use case diagram of the NL ANSProxy

A simplified class diagram of the NL ANSProxy is shown in Figure 10.

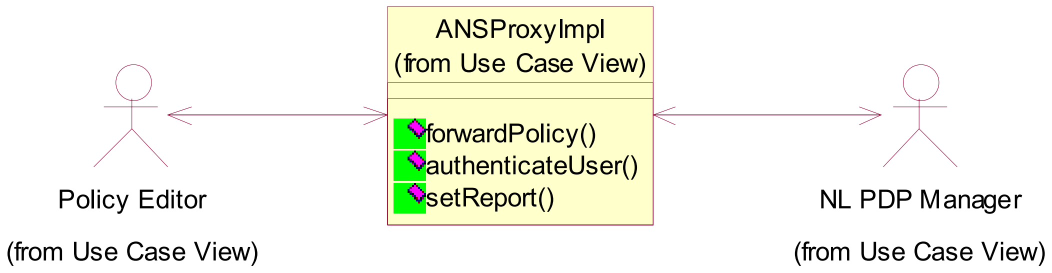


Figure 10. Simplified class diagram of the NL ANSProxy

The sequence diagram for the ANSProxy is shown in Figure 11.

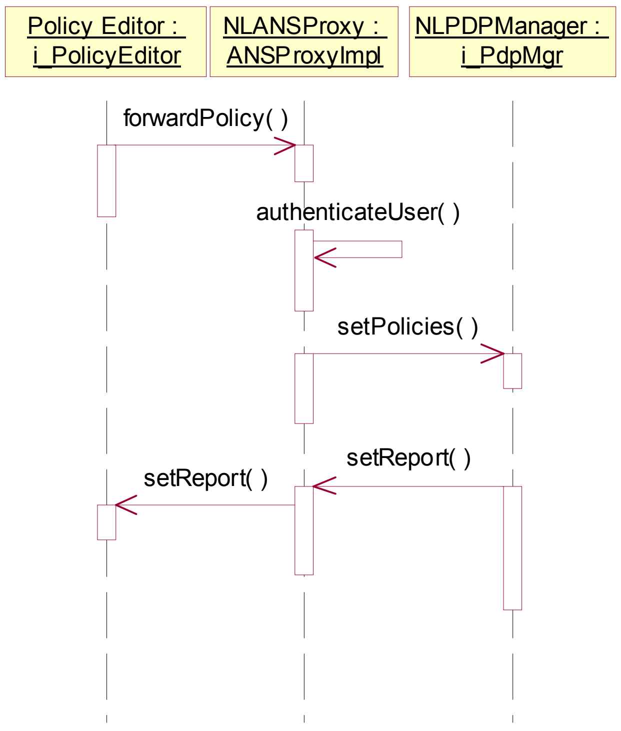


Figure 11. Sequence diagram of the NL ANSProxy

2.2.2.2 PDP Manager

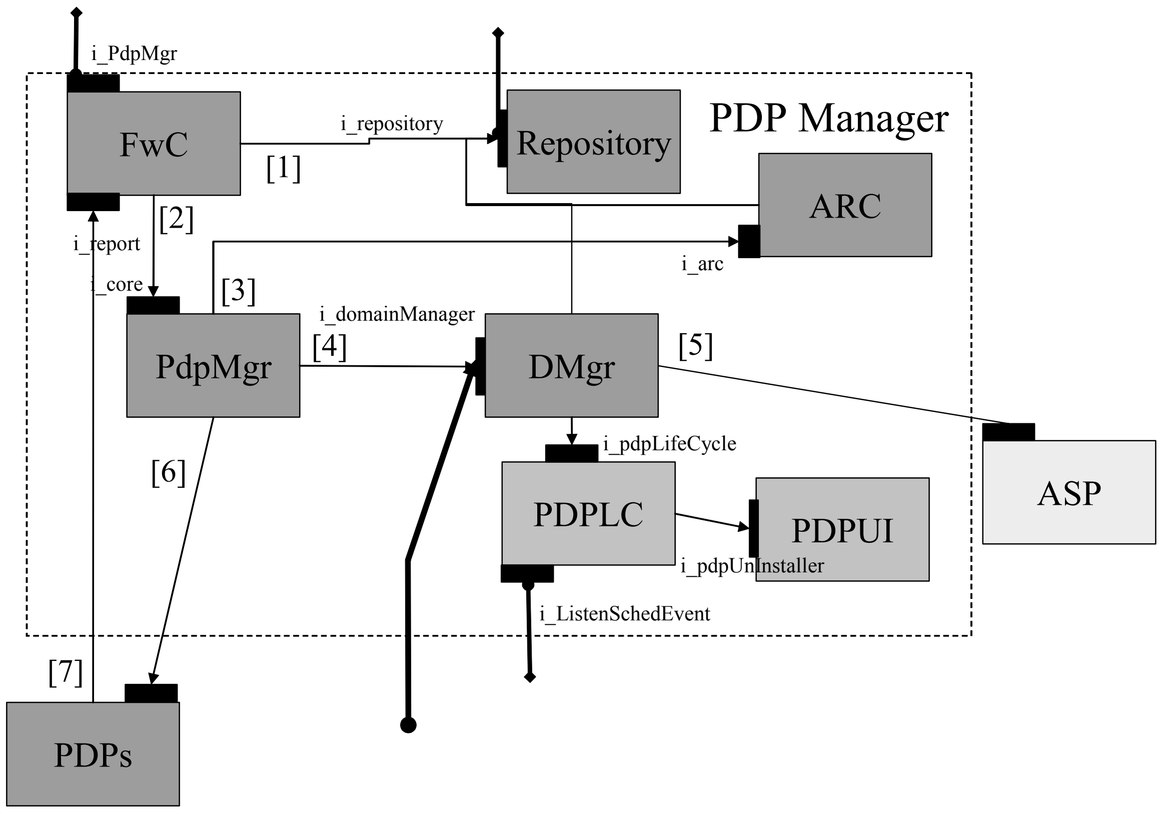


Figure 12. An Architectural Model for PDPManager

The PDP Manager is responsible for forwarding received policies to the appropriate Policy Decision Point (PDP). If the corresponding PDP is not installed, the PDP Manager requests the ASP system to download and install it, thereby extending the management functionality of the system as required. The sequence diagram in Figure 13 illustrates how the aforementioned extension is achieved.

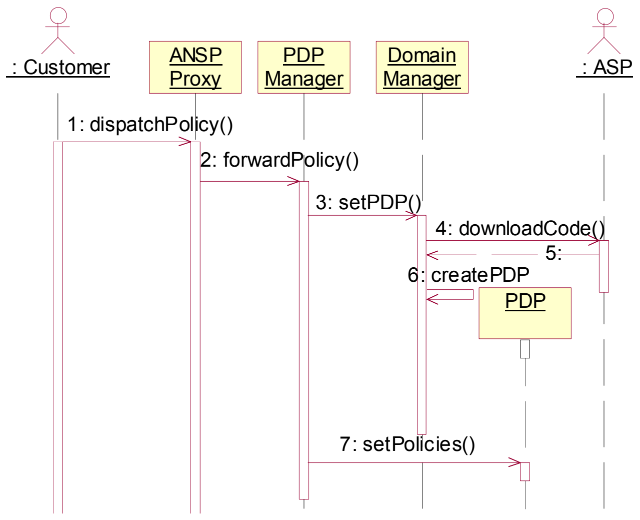


Figure 13. Dynamic Installation of a PDP

The Domain Manager (DMgr), a sub-component of the PDP Manager, is responsible for interacting with the ASP and instantiating the new PDP. Once the new PDP is deployed, the PDP Manager forwards the policy to it.

The PDP Manager also acts as a control point, as it has all the necessary information to understand the policy processing state. As an example, imagine that two different policies must be deployed but the second is only deployed if the first is successfully enforced. In this case, the PDP Manager keeps the second one in a halt state, until it receives notification of the first policy's successful enforcement.

Such a situation occurs, for instance when an SP requests the instantiation of a new virtual network and its corresponding Management Instance (MI). In this case, the PDP Manager receives two different types of policy, the QoS policy and the Access Rights Delegation policy. Following the described procedure it then installs the QoS policy (an action that requires admission control) and only when there are sufficient available resources (it receives the success notification) does it attempt to install the delegation policy. Only when both installations are completed successfully, does it instantiate the new MI and hand it over to the SP. Again, the entity responsible for the instantiation of the new MI is the Domain Manager.

Figure 14 shows the main components that form the PDP Manager and how they are inter-connected. The numbers show the flow that an incoming request follows.

When a policy is received: The ForwardController (FwC) (1) stores it in the repository before forwarding (2) it to the PDPMgr component. The latter will: first ask (3) the ARC component to check the access rights of the actor that sent the request; secondly, it requests (4) from the Domain Manager (DMgr) the reference of the PDP responsible for processing the policy. As we have mentioned above, the Domain Manager checks if this PDP is already running in the system and if not, it will perform the requisite installation. If the code required is not found locally, the Domain Manager contacts (5) the ASP to download it into the local cache. The installation procedure is then resumed. Once the new PDP is up and running, its reference is stored and returned back to the PDPMgr, who will use it to dispatch (6) the policy to the referenced PDP.

The PDP uses (7) the "i_report" interface to report the policy enforcement status of the policies dispatched to it. Then, the FwC component locally updates the status of the policy and checks whether there are more policies belonging to the same policy group waiting to be deployed. If so, then it restarts the process and notifies the PDP.

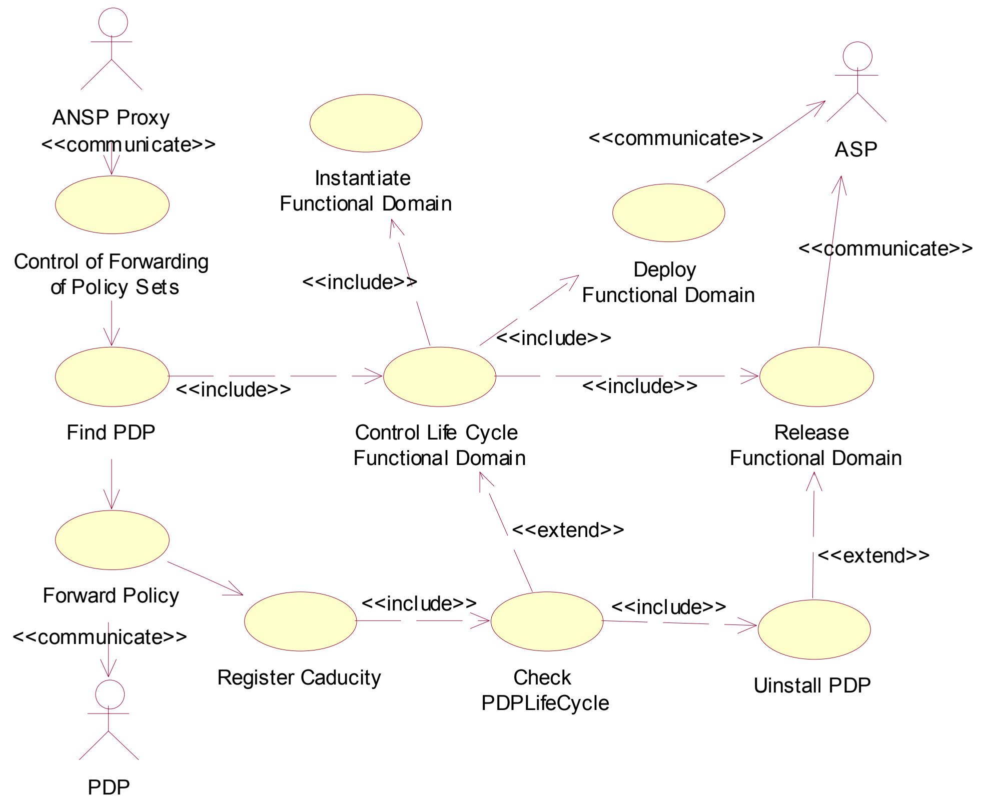


Figure 14. Use Cases for PDP Manager

Figure 14 illustrates the main use cases of the PDP Manager component. The mapping of these use cases with sub-components is:

The "Forward Controller" (FwC) component implements the functionality of the *"control of forwarding of policy sets"* use case. When a policy arrives, it checks if it is single policy or a policy set. In the case of a policy set FwC splits the set into individual policies, obtains the set forwarding mode and forwards the individual policies accordingly.

The "PdpMgr" sub-component co-ordinates the core behaviour of PDP Manager. It uses the Domain Manager component (DMgr) to realise the *"findPDP"* use case and, if the PDP is not found, it requests its installation to the DomainManager. The PdpMgr sub-component is also responsible for forwarding the policy to the PDP once it is installed. Then, it realises the register caducity use case using the Policy Lifecycle (PDPLC) sub-component.

The "Policy Lifecycle" (PDPLC) sub-component implements the functionality of the *"check PDP Life Cycle"* use case. It periodically checks if a PDP has expired. If so, it contacts the "PDP Uninstaller" in order to remove it, so fulfilling the *"uninstall PDP"* use case.

The DomainManager sub-component implements the functionality of the *"findPDP","Control LifeCycle Functional Domain", "Deploy Functional Domain" and "Release Functional"* use cases. The DomainManager sub-component allows the management framework to dynamically upgrade itself with new management capabilities. That is, it allows the management system to extend itself by downloading and creating an instance of a new functional domain.

The uses cases realized by this component are:

Find PDP: As previously described, when “PDP Manager” wants to retrieve the reference of a specific PDP, which belongs to a particular functional domain, it asks the DomainManager for this reference. Then the DomainManager looks in the local cache to see if the PDP is running on the system. If so, it returns its reference (IOR of the PDP), otherwise, it tries to create a new instance. If the code is already downloaded into the “classpath” then it returns a reference to the new instance, but if it does not have this code (it is not in the “classpath”), then it requests the ASP to initiate deployment of the code that implements the functional domain requested. Once the ASP has downloaded the code requested, it notifies the DomainManager which then creates a new instance and returns the reference of the newly instantiated PDP to the PDP Manager.

Control lifecycle Functional Domain: The DomainManager is responsible for triggering deployment of functional domains using “Deploy Functional Domain”. It also maintains the references of the currently instantiated functional domains, and provides these references to surrounding components, which use them to access the different components that comprise the functional domain (PDP/PEP).

Instantiate Functional Domain: A functional domain can be composed of more than one component, e.g. it can be composed of two components a PDP and a PEP., Currently the PDP is instantiated inside the management station (NMS, EMS). The PEP could be instantiated in the management station or in a VE inside a FAIN Active Node. When in the Active Node, the ASP system is responsible for instantiating the component.

Deploy Functional Domain: As already explained, when a PDP or PEP is not found locally, the DomainManager obtains from the ASP system the corresponding code. The policy has enough information to identify the functional domain responsible for processing the incoming request. This information is used to determine if the particular functional domain components are already running in the system or instead, they are stored locally in the management station.

Release Functional Domain: When a functional domain will not be used anymore it needs to be removed. The DomainManager is responsible for doing that. It will deactivate all those components that compose the functional domain. It requests the removal of PDPs and PEPs instantiated in the management system by contacting the “PDPLC” sub-component. If a PEP is located on a FAIN Active Node, it contacts the ASP to remove it.

2.2.2.3 PDP

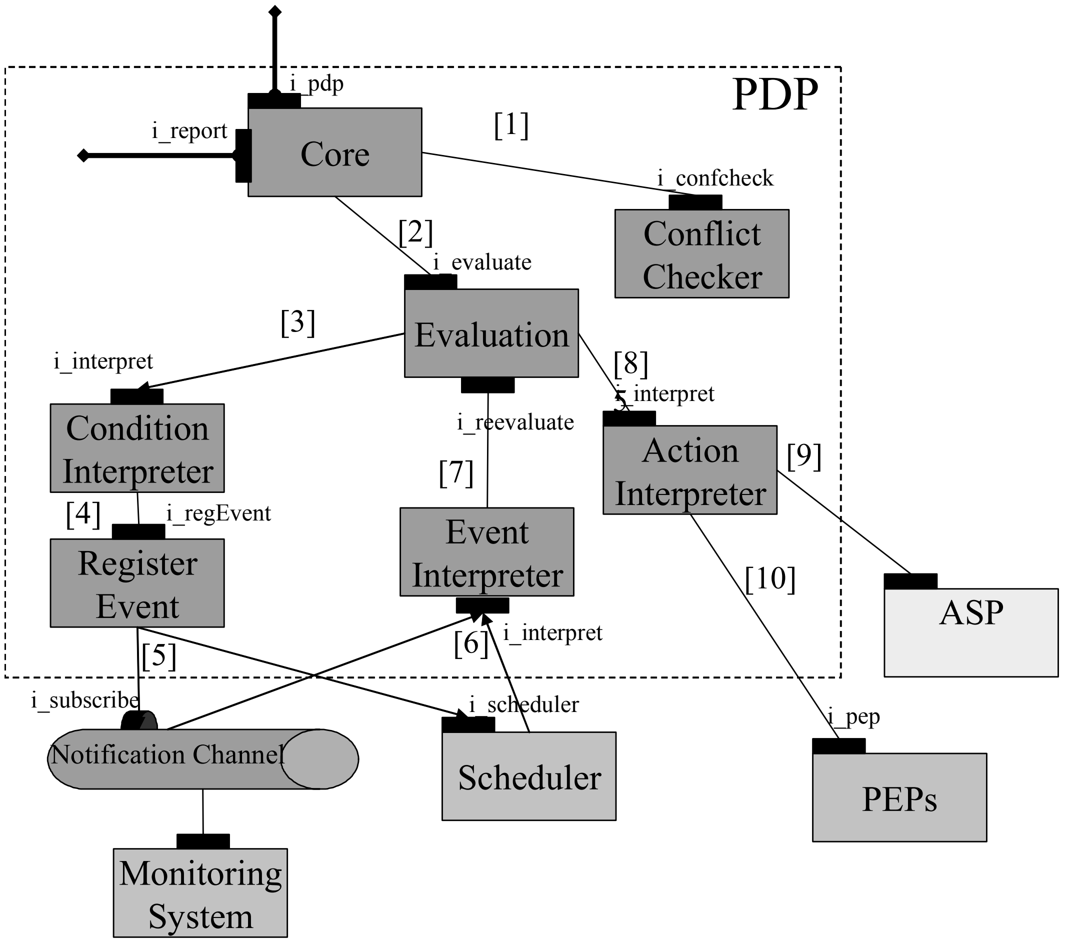


Figure 15. Architectural Computational Model for PDP

The FAIN management architecture accommodates different types of PDP, each one making decisions that apply to a specific functional domain, namely QoS PDP, Delegation of Access Rights PDP and Service-specific PDPs. They all perform conflict checks that are meaningful within their decision context (intra-PDP). In order to reach a decision, they also interact with other components that assist the PDPs in making a decision, e.g. a Resource Manager for admission control.

PDP is the main component in policy-based management architecture. Its main functionality is to check for possible syntactic and semantics conflicts in policies (sometimes, even try to solve these conflicts). Another role of the PDP is to decide when a policy should be enforced, for which purpose the PDP needs to receive information from the monitoring system. The third important function is to forward decisions to PEP components for enforcement.

Figure 16 illustrates the main use cases of a PDP component. The darker use cases should be extended and specialized for each management level.

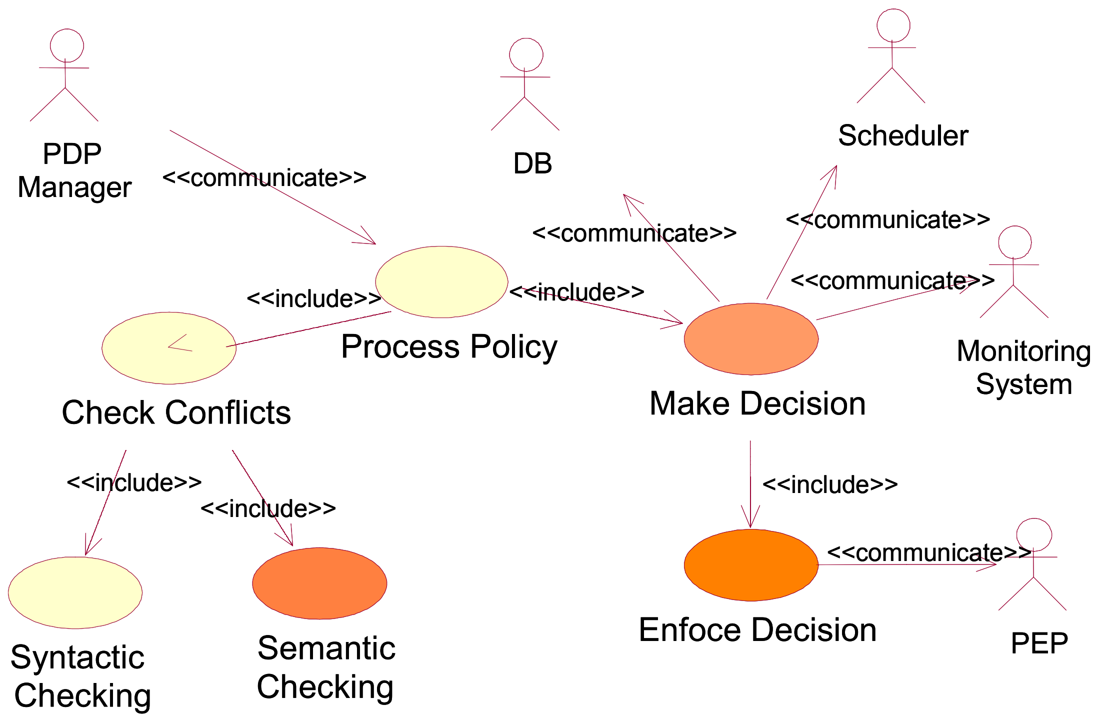


Figure 16. Use Case Diagram for Core PDP

The numbers shown in Figure 16 illustrate the processing logic associated with the deployment of a policy inside the PDP. All new incoming requests are received by the Core component, which is responsible for (1) checking if the enforcement of the policy will conflict with policies already enforced and running in the system. If there are no conflicts (2) the Core component sends the policy to be evaluated. The Evaluation engine analyses the policy, asking (3) the Condition Interpreter if the policy must be enforced immediately or whether it must wait until all the conditions evaluate as "true". In the case that some additional information is needed in order to take the decision, the Condition Interpreter (5) will either configure the Scheduler (if it is a time condition) or the Event Register (in case monitoring information is needed) and the Evaluation component will store the policy in the database. The Event Interpreter (6) receives all registered events either from the Notification Channel or from the Scheduler and contacts the Evaluation component (7) to re-evaluate the affected policies, and the whole process starts again. When all conditions are met, the Evaluation Engine initiates enforcement of the appropriate policies by requesting (8) the Action Interpreter Component to execute them and by sending the decision (10) to the corresponding PEP so that they are enforced.

Each PDP contains at least two types of component: the condition and action interpreters. These components provide action and condition processing logic for those policy types that are handled by the PDP. Each PDP has at least one instance of each type but they can be dynamically extended to accommodate more interpreters capable of processing new actions and conditions conveyed by the policies. The generic Condition and Action Interpreter make use of a particular field, inside the policy, to decide which particular action and condition interpreter should be used to process it. This particular processing logic is specific for each policy and hence for each functional domain and for each management level, i.e. there are condition/action interpreters specific to the element and network levels.

Figure 17 illustrates the sequence of events that take place when a new action interpreter is deployed. A generic Action Interpreter, acting as manager, becomes the recipient of all action requests carried by the policy. If there is an appropriate action interpreter already deployed in the PDP, the generic Action Interpreter forwards the request for further processing. Otherwise, it contacts the ASP in order to retrieve the action interpreter capable of processing the particular request.

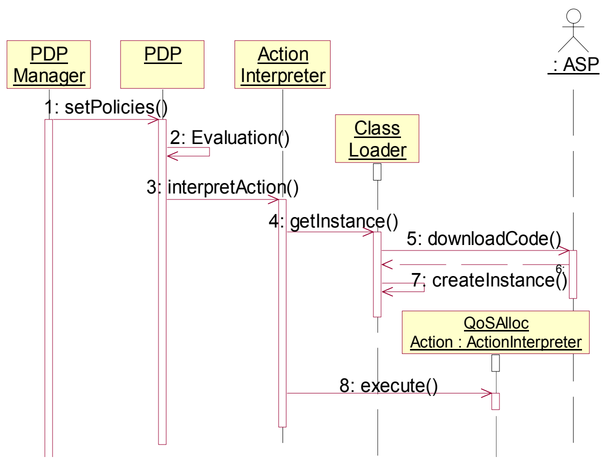


Figure 17. Dynamic Installation of an Action Interpreter

2.2.2.4 Monitoring System

Policy decisions rely on both local and global network status information. While PEPs are unsurpassable sources of device-specific data, a monitoring system is required to provide an overall picture of the network state. Obtaining such a picture in an Active Network, where new modules, service components and resource abstractions are constantly incorporated, is a challenge in terms of extensibility, scalability and efficiency.

These properties are architecturally addressed within the monitoring system in various sub-systems on both the network and element-levels. Indeed, grouping these trends in terms of responsibility led to the adoption of a layered architecture as shown in Figure 18. The three layers reflect the different aspects of the monitoring activity: while the acquisition layer gathers and processes data coming from network entities (offered by the active nodes through resource abstraction interfaces), the distribution layer permits an efficient delivery of such information to the PDPs through an extended notification channel; thirdly, the policy-based control layer aims to make decisions affecting the way the monitoring operations are carried out.

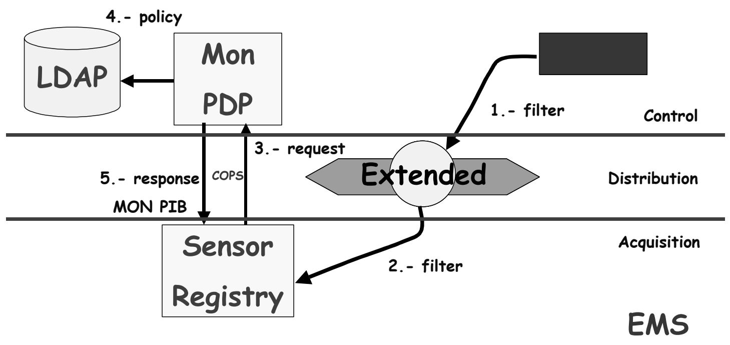


Figure 18. The FAIN Monitoring System Architecture

Altogether, these layers apply a set of strategies that guarantee an immediate response either to the appearance of new network elements or the need for new information processing methods, making the monitoring system inherently extensible. Such strategies roughly pertain either to the group of interface decoupling techniques or the application of the building block concept.

The first of these strategies, applied within the acquisition layer, relies on dynamically discovering the interfaces of the target to be monitored and analysing how to access them. Then, a set of parameters contained within runtime information (provisioned by the PDPs) is used to set up the monitoring operation. Even external metering blocks are subjected to this type of configuration. This approach makes it feasible to access components that were not initially foreseen and immediately extends the policy-based configuration mechanisms to virtually any measurement element.

Dynamic analysis techniques are also applied to the events produced by event sources. Their structures are traversed and the information they contain is reorganized for its automatic adaptation and delivery as structured events, since their fields thus become capable of further filtering evaluation within the notification channel.

In higher abstraction layers, the use of an event channel as the only means to interchange both events and configuration orders (embedded in filtering constraints) guarantees interface independence between the PDPs and the monitoring system.

Extensibility is also pursued by applying the building block concept to design the data processing elements. Firstly, a set of basic and generic data manipulation blocks have been defined (threshold surveillance, statistics blocks, etc). Secondly, appropriate rules for connecting such blocks into manipulation chains have been specified. Finally, the configuration of the manipulation chains is flexibly defined in monitoring policies. Altogether, these techniques are used to create new data processing functionalities.

Flexibility is achieved through special features of the distribution layer. Through the use of a notification channel, this layer decouples the PDPs from the network entities that generate the events. Furthermore, the distribution layer ensures that events remain within limits. This feature and the hierarchical arrangement of notification channels within the FAIN network promote scalability by preventing unsolicited element-level events from reaching the network-level management PDPs.

2.2.2.5 Policy Parser

The policy parser converts XML encoded policies into their Java component counterparts. It provides mechanisms for ensuring policy correctness and enables automatic generation of the XML code corresponding to the different policy elements. The parser design distinguishes a set of structural classes and a set of classes actually holding the information. The former provide the connection points between the diverse information elements whereas the latter offer those fields defined in the XML policy schemas, through their respective interfaces**.**

One of the major features of this design is its extensibility. The structural classes contain unmarshalling methods, which allow dynamic population of the policy structures with information classes not even foreseen at design time. The information class lookup is performed based on the data held in the XML policy itself, and is therefore a self-sufficient mechanism. Extensibility is also improved by the fact that each class holds its own marshalling and unmarshalling code, so that each new class embodies an "ad hoc" parser, able to interpret the corresponding XML document.

In order to keep the design sufficiently simple and robust, only two main API subsets have been defined: those accessor methods for retrieving and storing the policy information and those methods for realizing the marshalling and unmarshalling procedures.

Nevertheless, after analysing the API from the user perspective, it has been found useful to provide an additional interface for performing the direct parsing of a complete XML document, making use of the methods offered by the rest of policy classes. This entry point facilitates the correct use of the API, completely isolating the applications from the XML document management.

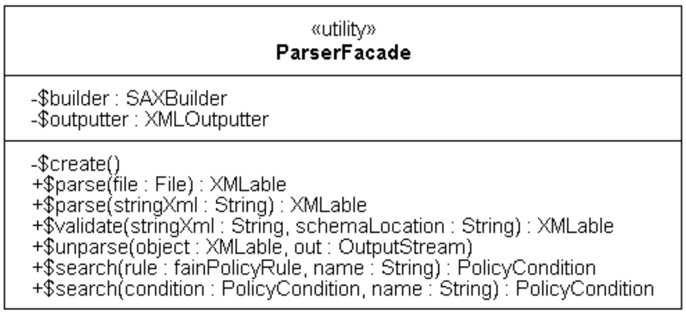

«utility»
ParserFacade
-$builder : SAXBuilder
-$outputter : XMLOutputter
-$create()
+$parse(file : File) : XMLable
+$parse(stringXml : String) : XMLable
+$validate(stringXml : String, schemaLocation : String) : XMLable
+$unparse(object : XMLable, out : OutputStream)
+$search(rule : fainPolicyRule, name : String) : PolicyCondition
+$search(condition : PolicyCondition, name : String) : PolicyCondition

Figure 19. Parser facade class diagram

The parser provides an additional easy-to-use method for validating the XML policy against a specified schema. Simplifying the use of the parser has been a major design goal that led to the definition of an API which reproduces the fields defined in the XML schema. The use of accessor methods and the implementation of the serializable interface favour the dynamic discovery of the policy properties using introspection mechanisms.

Figure 20 displays the main relationships between the classes involved in supporting the policy information. Each policy holds a list of ConditionReferences and ActionReferences as well as the set of attributes defined in the IETF specifications. The ConditionReferences and ActionReferences point to PolicyConditions and PolicyActions respectively, thus acting as general containers. Each policy class maintains a JDOM eElement as a private attribute, which is joined to the particular part of the main XML document associated to the policy.

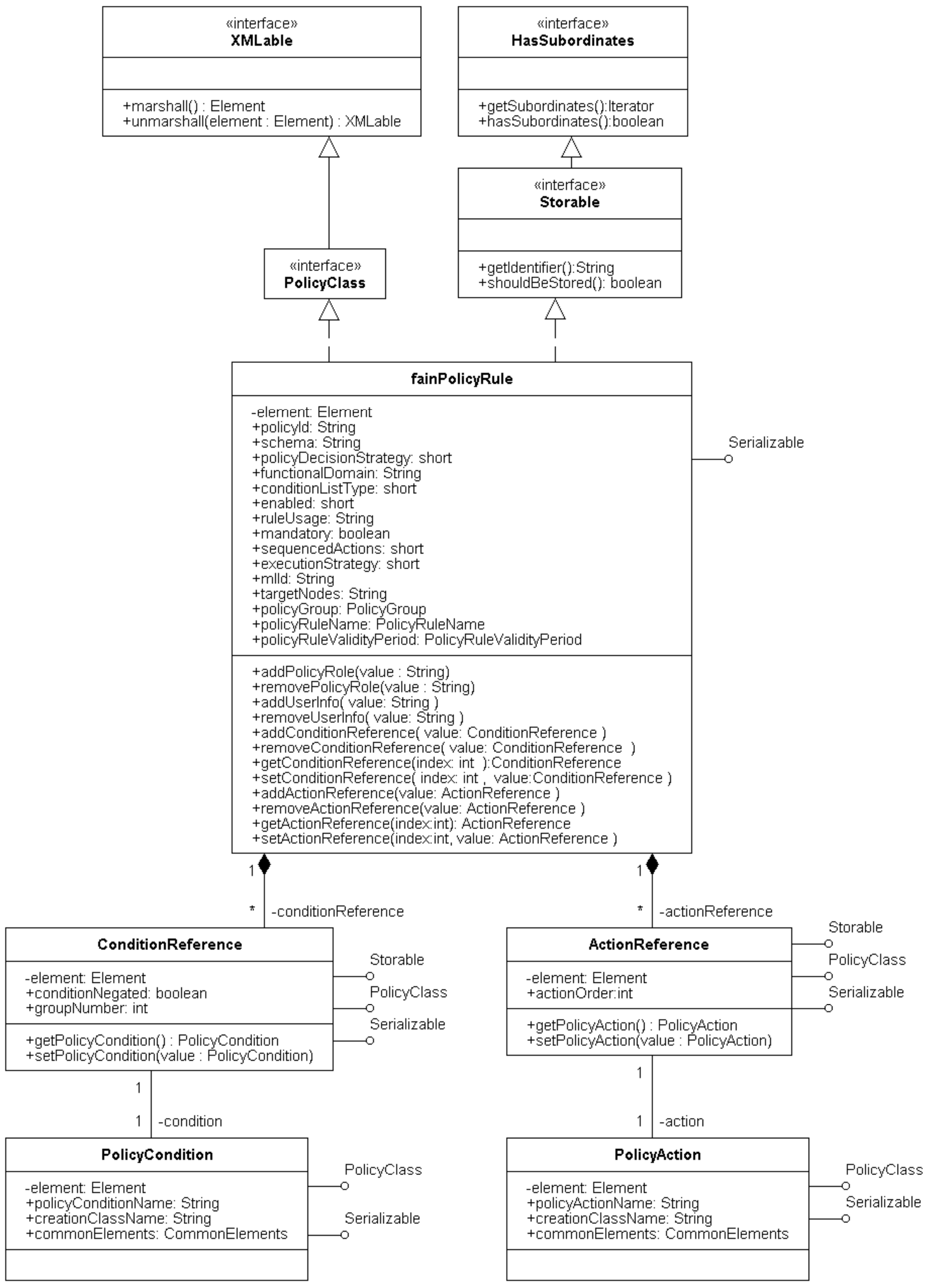

«interface»
XMLable
+marshall() : Element
+unmarshall(element : Element) : XMLable
«interface»
HasSubordinates
+getSubordinates():Iterator
+hasSubordinates():boolean
«interface»
Storable
+getIdentifier():String
+shouldBeStored(): boolean
«interface»
PolicyClass
fainPolicyRule
-element: Element
+policyId: String
+schema: String
+policyDecisionStrategy: short
+functionalDomain: String
+conditionListType: short
+enabled: short
+ruleUsage: String
+mandatory: boolean
+sequencedActions: short
+executionStrategy: short
+mlId: String
+targetNodes: String
+policyGroup: PolicyGroup
+policyRuleName: PolicyRuleName
+policyRuleValidityPeriod: PolicyRuleValidityPeriod
+addPolicyRole(value : String)
+removePolicyRole(value : String)
+addUserInfo(value: String)
+removeUserInfo(value: String)
+addConditionReference(value: ConditionReference)
+removeConditionReference(value: ConditionReference)
+getConditionReference(index: int):ConditionReference
+setConditionReference(index: int , value:ConditionReference)
+addActionReference(value: ActionReference)
+removeActionReference(value: ActionReference)
+getActionReference(index:int): ActionReference
+setActionReference(index:int, value: ActionReference)
Serializable
1
*
-conditionReference
1
*
-actionReference
ConditionReference
-element: Element
+conditionNegated: boolean
+groupNumber: int
+getPolicyCondition() : PolicyCondition
+setPolicyCondition(value : PolicyCondition)
Storable
PolicyClass
Serializable
ActionReference
-element: Element
+actionOrder:int
+getPolicyAction() : PolicyAction
+setPolicyAction(value : PolicyAction)
Storable
PolicyClass
Serializable
1
1
-condition
1
1
-action
PolicyCondition
-element: Element
+policyConditionName: String
+creationClassName: String
+commonElements: CommonElements
PolicyClass
Serializable
PolicyAction
-element: Element
+policyActionName: String
+creationClassName: String
+commonElements: CommonElements
PolicyClass
Serializable

Figure 20. Policy Components Class Diagram

The policy parser maintains an updated copy of the XML document, minimizing the amount of memory required to hold the data field values. This strategy also benefits the overall performance in two ways: firstly, the unmarshalling operations are delayed until the moment the information is actually required;. secondly, the marshalling procedures are carried out as soon as the information is available which decreases the time required to provide the complete XML document once requested.

Finally, the parser classes implement the storable interface defined as part of the database package. Through this interface, the classes hierarchical relationship information is provided to the policy database controller in a flexible way.

Different types of policy conditions are represented as sub-classes of the PolicyCondition, as shown in Figure 21. The creation of complex policy conditions is automatically managed by the CompoundFilterConditions, which finally store the resulting set of ConditionsReferences. An actual implementation of conditions is provided by the SimplePolicyConditions, which maintain the relationship between PolicyVariables and a PolicyValues.

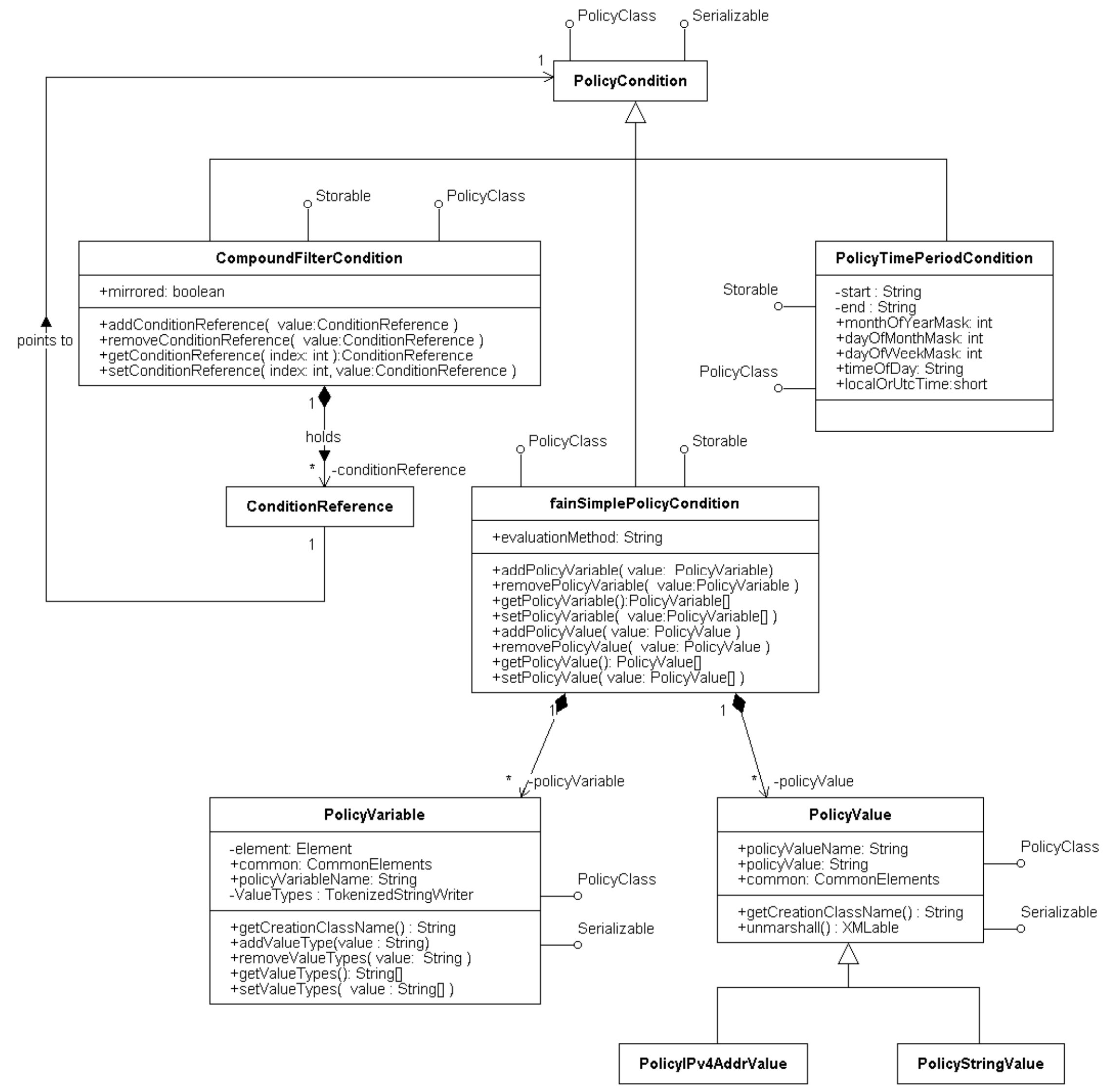

PolicyClass
Serializable
1
PolicyCondition
Storable
PolicyClass
CompoundFilterCondition
+mirrored: boolean
+addConditionReference(value:ConditionReference)
+removeConditionReference(value:ConditionReference)
+getConditionReference(index: int):ConditionReference
+setConditionReference(index: int, value:ConditionReference)
points to
1
holds
*
-conditionReference
ConditionReference
1
PolicyTimePeriodCondition
Storable
PolicyClass
-start : String
-end : String
+monthOfYearMask: int
+dayOfMonthMask: int
+dayOfWeekMask: int
+timeOfDay: String
+localOrUtcTime:short
PolicyClass
Storable
fainSimplePolicyCondition
+evaluationMethod: String
+addPolicyVariable(value: PolicyVariable)
+removePolicyVariable(value:PolicyVariable)
+getPolicyVariable():PolicyVariable[]
+setPolicyVariable(value:PolicyVariable[])
+addPolicyValue(value: PolicyValue)
+removePolicyValue(value: PolicyValue)
+getPolicyValue(): PolicyValue[]
+setPolicyValue(value: PolicyValue[])
1
1
*
-policyVariable
*
-policyValue
PolicyVariable
-element: Element
+common: CommonElements
+policyVariableName: String
-ValueTypes : TokenizedStringWriter
+getCreationClassName() : String
+addValueType(value : String)
+removeValueTypes(value: String)
+getValueTypes(): String[]
+setValueTypes(value : String[])
PolicyClass
Serializable
PolicyValue
+policyValueName: String
+policyValue: String
+common: CommonElements
+getCreationClassName() : String
+unmarshall() : XMLable
PolicyClass
Serializable
PolicyIPv4AddrValue
PolicyStringValue

Figure 21. Policy Condition Class Diagram

PolicyActions follow a similar approach. Actions that are specific to different technology domains extend the fainSimplePolicyAction, which includes basic code for storing and manipulating the JDOM eElement. The following figure is a simplified diagram of the classes involved in supporting policy action operations.

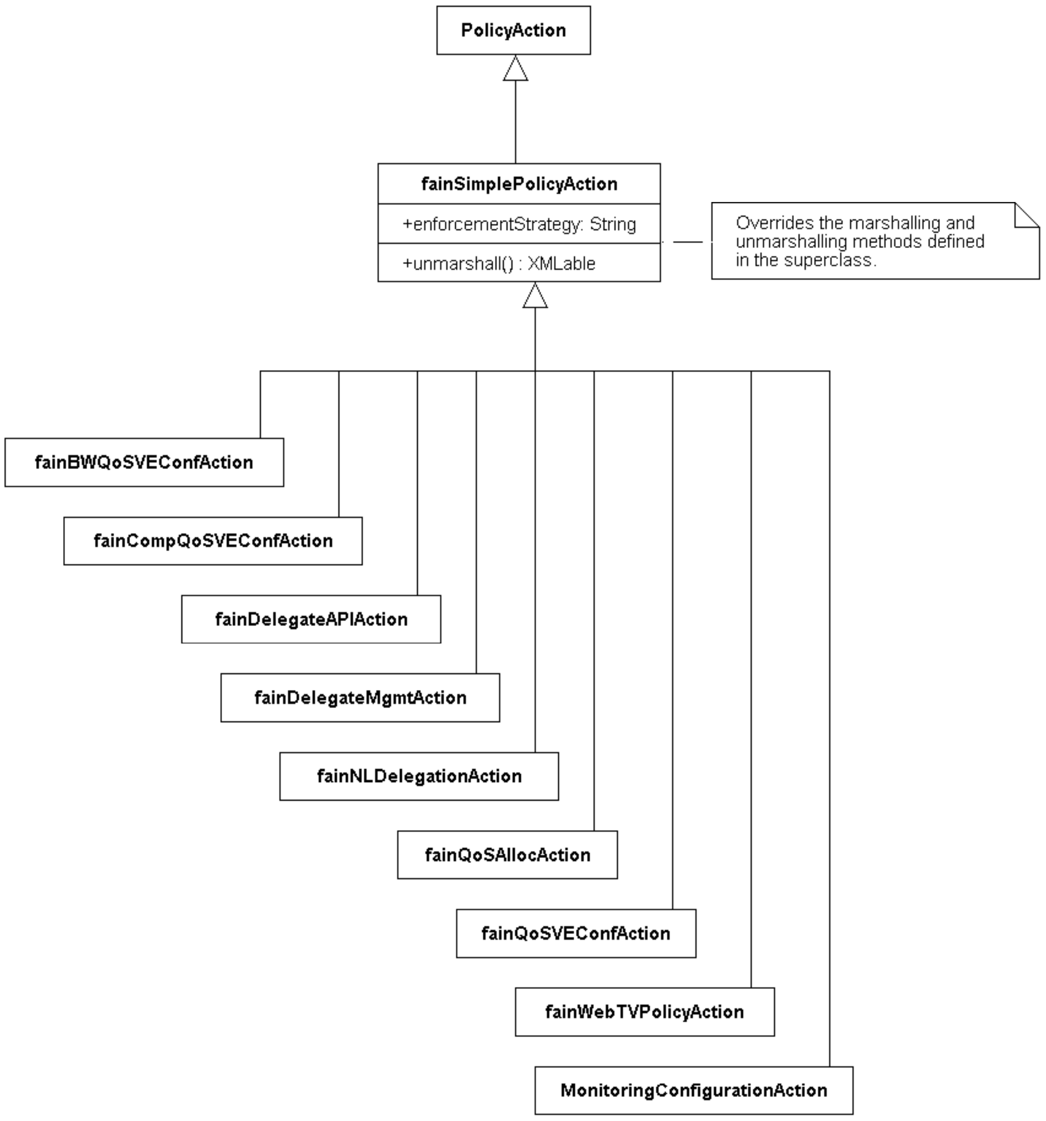


Figure 22. Policy Action Class Diagram

2.2.2.6 *Policy Repository*

The policy repository is supported on a LDAP directory, which provides content-based policy searches and distribution transparency. These features make it suitable for providing scalable storage solutions in network wide systems, such as the fain management system.

In general, accessing LDAP directories from Java applications is enabled through the use of the JNDI (Java Naming and Directory Interface) API. JNDI offers an abstract view of the LDAP directory, hiding the LDAP specific operations under a standardised interface suitable for interacting with different storage services that follow a similar approach. The following diagram illustrates the basic blocks building the repository.

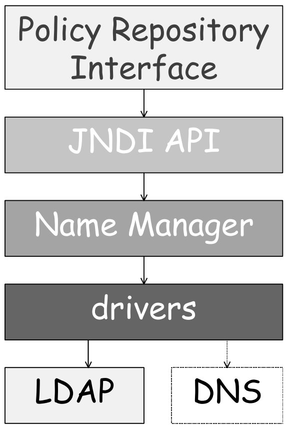


Figure 23. Policy Repository Access Interface Structure

The LDAP directory might be accessed directly from the PDP internal components or through an intermediate cache that would improve the overall efficiency.

A reduced set of requirements have determined the policy repository design, namely:

- Access to the repository shall be technology transparent. Neither LDAP nor JNDI specific issues shall be exposed to the outside components visiting the policy database.
- A centralized component (controller) shall organise the directory look-ups.
- The policy directory shall provide searching mechanisms based on policy, condition or action specific attributes.

As a design decision to enhance performance, only essential look-up attributes are stored in the directory. Since only java applications are intended to access the directory, the impact of such decision on interoperability is limited.

The policy directory basically consists of a database access controller and a series of state factories and object factories appropriate for the different policy classes. As stated in [1], the database access controller provides a simple query interface for efficient database searching operations. The main responsibility of the controller is to "locate the requested policies, retrieve them and return them in an appropriate format". The controller is also in charge of directing the storage process according to the specified hierarchical relationships.

The state factories and objects factories are merely format translators that convert java objects into LDAP entries and vice versa, respectively.

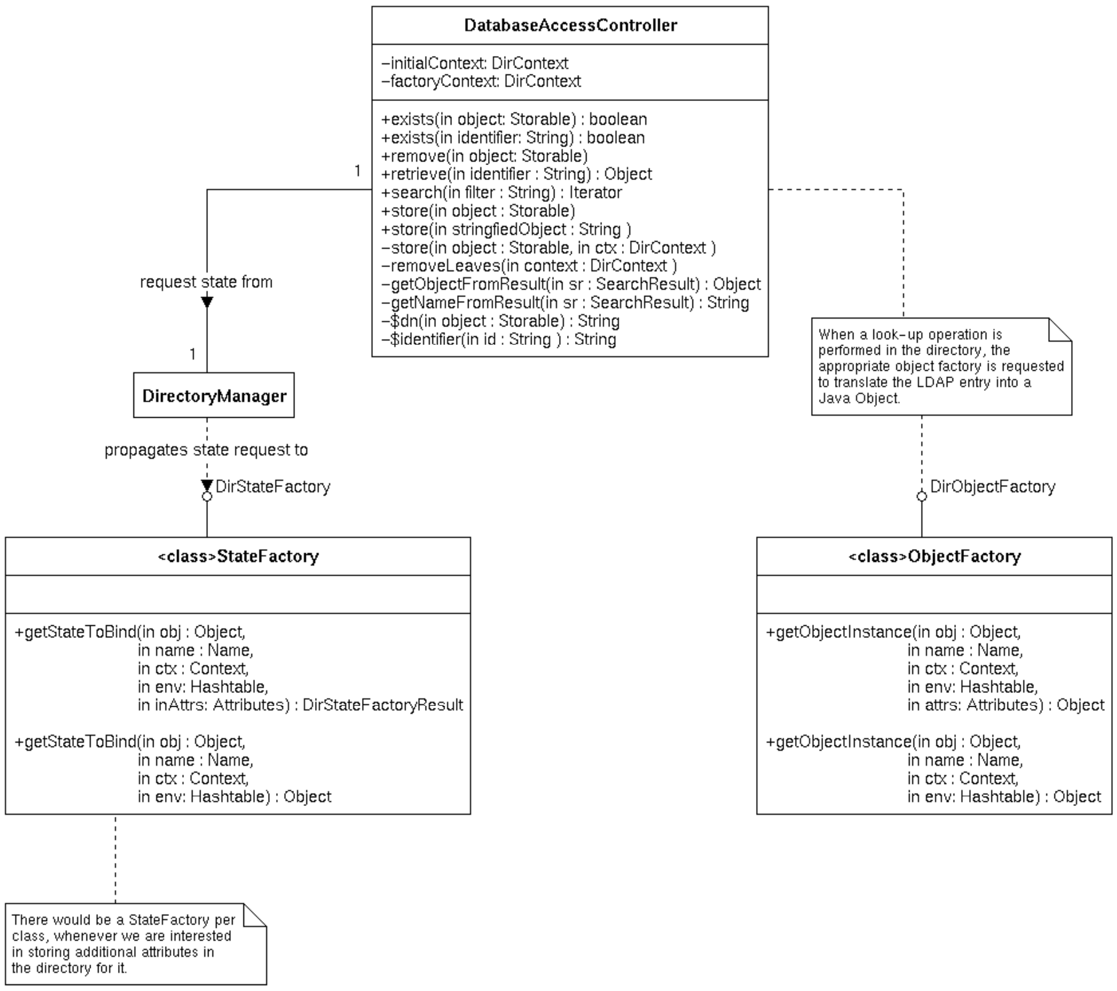

DatabaseAccessController
−initialContext: DirContext
−factoryContext: DirContext
+exists(in object: Storable) : boolean
+exists(in identifier: String) : boolean
+remove(in object: Storable)
+retrieve(in identifier : String) : Object
+search(in filter : String) : Iterator
+store(in object : Storable)
+store(in stringfiedObject : String)
−store(in object : Storable, in ctx : DirContext)
−removeLeaves(in context : DirContext)
−getObjectFromResult(in sr : SearchResult) : Object
−getNameFromResult(in sr : SearchResult) : String
−$dn(in object : Storable) : String
−$identifier(in id : String) : String
1
request state from
1
DirectoryManager
propagates state request to
DirStateFactory
When a look-up operation is performed in the directory, the appropriate object factory is requested to translate the LDAP entry into a Java Object.
DirObjectFactory
<class>StateFactory
+getStateToBind(in obj : Object, in name : Name, in ctx : Context, in env: Hashtable, in inAttrs: Attributes) : DirStateFactoryResult
+getStateToBind(in obj : Object, in name : Name, in ctx : Context, in env: Hashtable) : Object
<class>ObjectFactory
+getObjectInstance(in obj : Object, in name : Name, in ctx : Context, in env: Hashtable, in attrs: Attributes) : Object
+getObjectInstance(in obj : Object, in name : Name, in ctx : Context, in env: Hashtable) : Object
There would be a StateFactory per class, whenever we are interested in storing additional attributes in the directory for it.

Figure 24. Policy Repository Class Diagram

Each class that may be stored in the repository must implement the Storable interface that contains operations for providing the hierarchical information not contained in the schema, and its identifier (a distinguished name that must be obtained so that there is no collision when storing the entry in the directory).

The policy schema defines the type of policy classes that can be stored in the LDAP database and the valid attributes for each of them. The hierarchical relationships existing between the classes is not reflected in the schema but maintained on the directory structure itself (as in the case of a filesystem).

2.3. Network-level Management System (NMS)

Three additional components make the NMS distinct from the element-level management system: the Service Manager (SM), the Resource Manager (RM) and the Inter-Domain Manager (IDM) which support service deployment, decision-making with regards to resources control, and inter-domain communication respectively. The roles of these components deal with network-wide issues. The SM is responsible for setting up a VAN for a particular service used by a specific SP. Furthermore the SM is responsible for initiating the deployment of the service on the created VAN.

The RM component provides to PDPs the best domain wide route according to resource status. The inter-domain component is responsible for furnishing the mechanisms for communication with other domains. In the PBNM system, the inter-domain component is responsible for conveying and managing requests over FAIN domains. Figure 25 illustrates the management system's components and how SM, IDM and RM are integrated within the NMS.

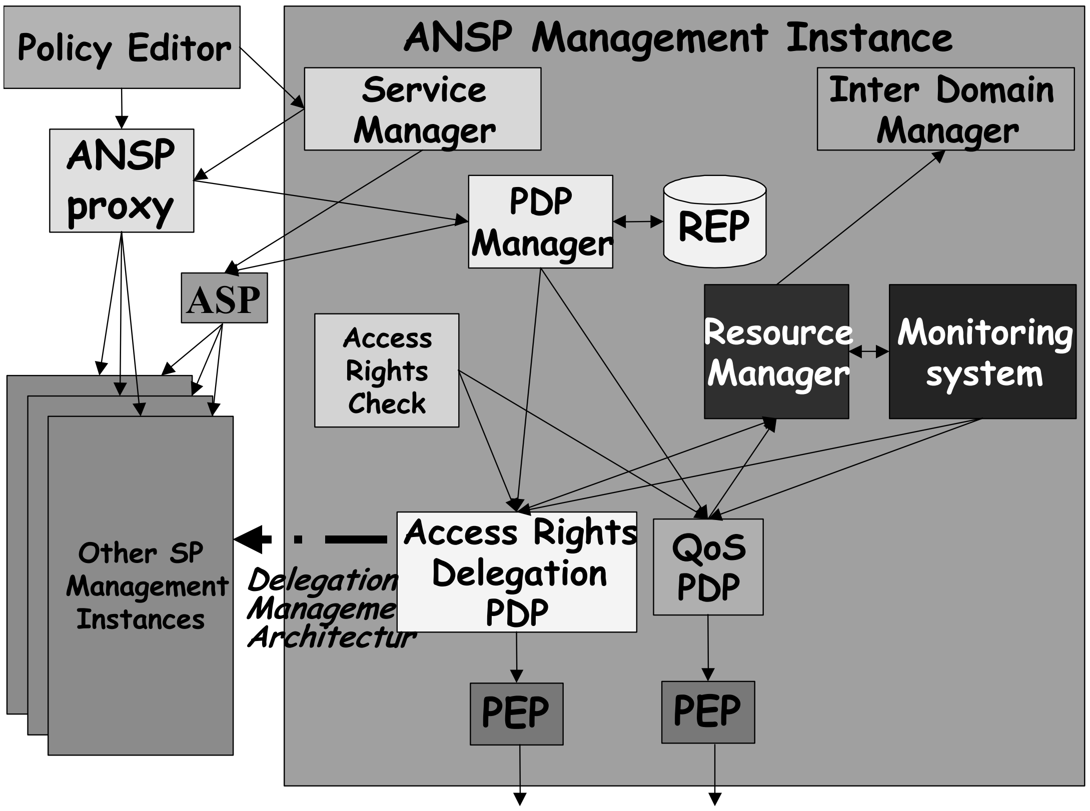


Figure 25. Architectural Model for NMS

2.3.1 Use cases

This section describes the main use cases, which are more closely related with the network level than with the element level. The use cases, which will be covered, are the delegate management functionality, the manage service, the inter-domain management, the use management instance and the signalling.

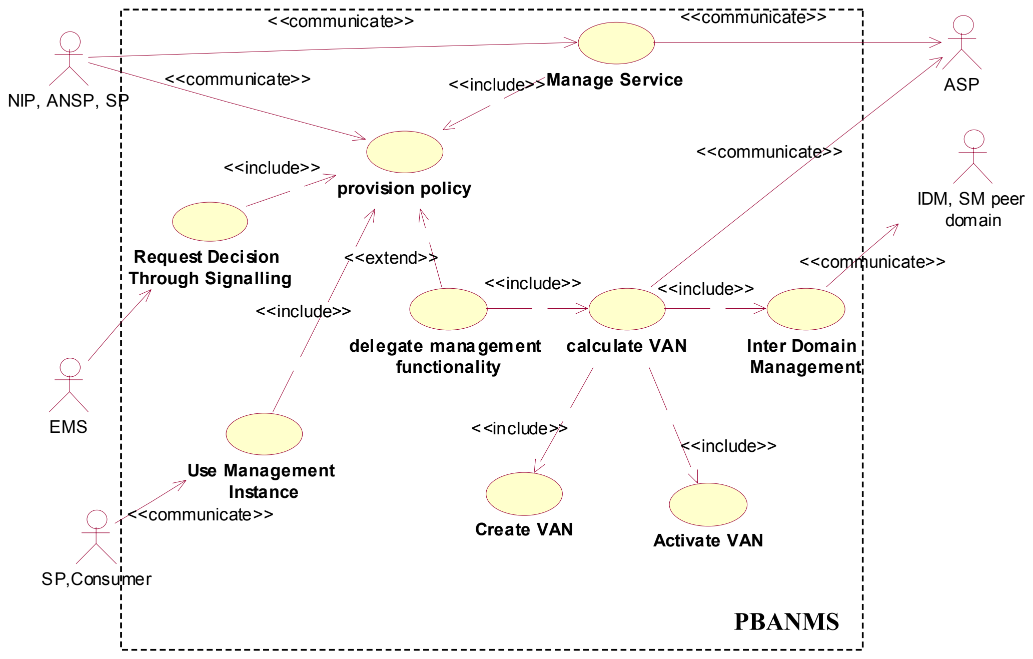


Figure 26. Use Cases for PBNM

Delegate Management Functionality: The delegate Management functionality use case is conceptually almost identical to the provision policy use case explained in the core use cases section. The only difference is that, in this case, the provisioning actions are the creation and activation of a Virtual Active Network (VAN) and a management instance for a new actor with certain access rights.

As previously described, most of the functions are the same as those defined for the provision policy use case. For this reason we will not repeat them here, but will simply.highlight the main features of the delegation of management functionality use case. As stated in the diagram, this use case can only be realised by the NIP, ANSP or SP actors, and not by the Consumer since it cannot delegate management functionality to any other actor.

Manage Service: A new function included within the FAIN management framework is reflected in the "Manage Service" use case. As stated in the use cases diagram NIP, ANSP and SP can in theory realise this use case each time the SP wants to deploy a service the Service Manager (this component will be describe later). SM will coordinate the creation of a VAN for deploying a given service as follows:

- Retrieve from ASP all topological requirements associated to the given service.
- Generate the appropriate policies for allocating resources along the VAN path to be created
- Generate the appropriate policies for delegating management functionality on the appropriate Virtual Environment nodes that constitutes the VAN.
- Communicate with ASP to trigger deployment of the given service within the already created/activated VAN.

Inter Domain Management: This is a new feature included within the FAIN management framework. It is mainly used to allow a FAIN ANSP to manage requests over FAIN domains. This happens when an SP wants to deploy a service over different administrative domains. For instance it allows the Resource Manager to provide the best route across different domains.

Use Management Instance: As stated in the use case diagram the SP and authorised Consumers can make use of the functionality described by the "provisioning policy" use case to manage the allocate resource and services.

Request decision through signalling: The signalling approach is another basic feature of a policy-based system; where the managed device (AN) requests through the policy enforcement point (EMS) a set of resources to the decision point (NMS). Depending on the available resources, and on the policies available in the system, the policy decision point decides whether this request should be accepted, and thus whether the resources are allocated or rejected.

2.3.2 Application Programming Interfaces (API)

We describe the main interfaces offered by the NMS to external sub-systems.

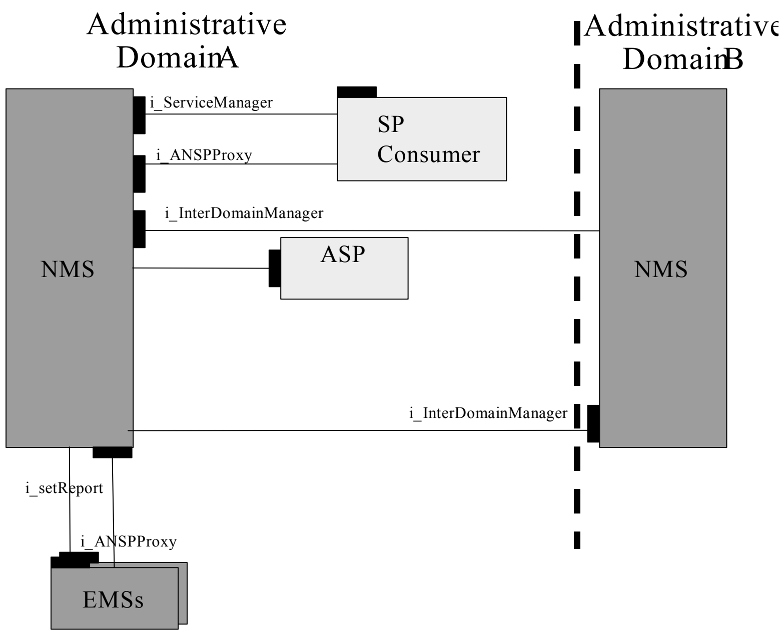


Figure 27. Interfaces offered by an NMS

i_ServiceManager: The SM component presents this interface to Consumers and SPs or their NMS, enabling them to initiate deployment of particular services within an administrative domain.

This is the IDL description of this interface:

```
    /** Service Deployment Wizard Interface.*/
     interface ServiceDeploymentWizard:editor::ReportConsumer{
    /** Triggers the creation of a VAN to afterwards deploy the  service 
given.*/
    network::asp::ServiceReferenceInfos deployService   (
                           in network::asp::ServiceName serviceName,
                           in network::VANNodes userNodes,
                           in tCredential credential
    ) raises((
network::asp::ServiceNotFound,
                           VANAllocationFailed,
                                  network::asp::InstantiationFailed,
                                  network::asp::ServiceComponentNotFound
    );
    ….
    };
```

Where:

Argument Name	Argument Type	Description
ServiceName	String	The name of the Service to deploy.
UserNodes	VANNodes	Array of VANNode. VANNode is a structure that contains the attributes of the sites to be interconnected. IDL type defined as follows: `struct VANNode { VANNodeID nodeID; Properties props; };`
Credential	tCredential	Structure that contains the attributes of the SP that request the action. IDL type defined as follows: `struct tCredential { string name; tOctetList key; };`
Returns	**ServiceReferenceInfos** Structure that contains handlers needed for accessing to the service deployed.IDL structure defined as follows: `typedef string ServiceComponentID; typedef string ServiceReference; struct ServiceReferenceInfo { ServiceComponentID componentID; ServiceReference serviceReference; };`	
Exceptions	**Description**	
Network::asp::ServiceNotFound	Raised when the Service Requested cannot be found on the net Service Registry.	
VANAllocationFailed,	Raised when the VAN cannot be created/activated due to a lack of resources.	
Network::asp::InstantiationFailed	Raised when there are any problems during the Instantiation of any component that forms the service.	
Network::asp::ServiceComponentNotFound	Raised when a service component	

i_ANSProxy: This interfaces is offered by the ANSProxy Component an is the initial access point for to SP, ANSP, Policy Editor, and SM. ANSProxy receives a request in form of a policy and then processes it.

```
   /** AnspProxy  Interface.*/
           interface iANSProxy {
             oneway void forwardPolicy((in string policy));
```

This is the IDL description for the interface:

Where:

Argument Name	Argument Type	Description
policy	String	String containing the incoming policy. The policy is defined following the PCIM Information Model
Returns	Nothing. It is an asynchronous call. A one-way CORBA communication type.	
Exceptions		

i_setReport: This interface is offered by all the PEPs and is used for the EMSs to send a report about the actual policy enforcement status.

This is the IDL description for the interface:

```
interface i_setReport {
          oneway void setReport(in t_Report report);
};
```

Where:

Argument Name	Argument Type	Description

Report	t_Report	Structure that contains attributes to show the actual policy status enforcement. IDL type defined as follows:
		`enum t_policyStatus { INPROGRESS, DONE, FAILED };`
		`struct t_Report {`
		`string owner;`
		`string pdpName;`
		`string policyRef;`
		`t_policyStatus status;`
		`string details;`
		`};`

Returns	Nothing. It is an asynchronous call. A one-way CORBA communication type.	
Exceptions		

2.3.3 NMS Components

2.3.3.1 Policy Editor

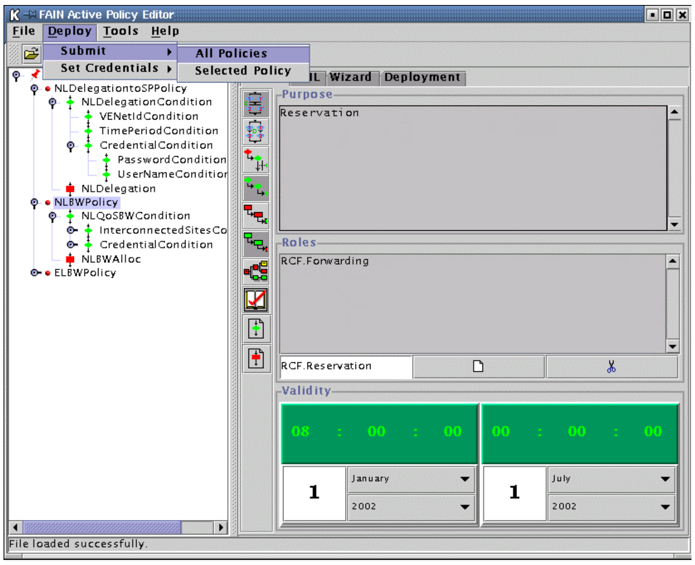
The policy editor permits creation and modification of management policies in a graphically assisted environment, as well as supervising the deployment of such policies within the network. To enhance the manipulation of the policy information, it incorporates interpreters enabling the translation of the XML structures into graphical elements that represent each of the policy components. Such graphical elements are then hierarchically arranged in a tree panel, providing the view of the policy layout.

In the policy view (Figure 28) it is possible to conductfine-grain operations: for example, selection of a tree element causes its associated attributes to be displayed in a property sheet, so that they can be viewed or changed. In the tree view, new elements may be added directly into the policy structure. This process is easily performed by selecting one of the available policy components (rule, condition or action) in the toolbar and subsequently clicking on the desired point in the tree.

Once the policy is considered to be complete, deployment is initiated from the policy editor menu. A validation process is automatically carried out before actually deploying the policy, and the user is informed of any problems. The policy editor then contacts the ANSProxy and forwards the resulting XML document through the i_ANSProxy interface.

During deployment, the editor gathers and displays the reports sent by the different entities involved in the enforcement chain, which allows detection and location of any fault or conflict that may arise, and facilitates its solution.

Figure 28 shows a snapshot of the policy editor with the main areas that have been formerly described.



Purpose
Reservation

Roles
RCF.Forwarding
RCF.Reservation

Validity
08 : 00 : 00 — 1 January 2002
00 : 00 : 00 — 1 July 2002

File loaded successfully.

Figure 28. Snapshot of the FAIN Policy Editor

2.3.3.2 Service Manager (SM)

The service manager (SM) is responsible for setting up a VAN for a particular SP and service. It receives (as input) the SLA (agreed between ANSP and SP) and sites to interconnect, as well as the services to deploy. We refer to the first input as a static requirement, while the second and third are known as context information, which are essentially dynamic requirements. The SM uses this information together with the topological service requirements, imposed by the service (if relevant), which are retrieved from the NetASP, to generate the appropriated set of NL QoS and Delegation of Access Rights policies. As a result of the enforcement of these policies a VAN is created and SM contacts with NetASP to trigger the deployment of the service on the VAN created.

The figure below illustrates the sequence of events explained above.

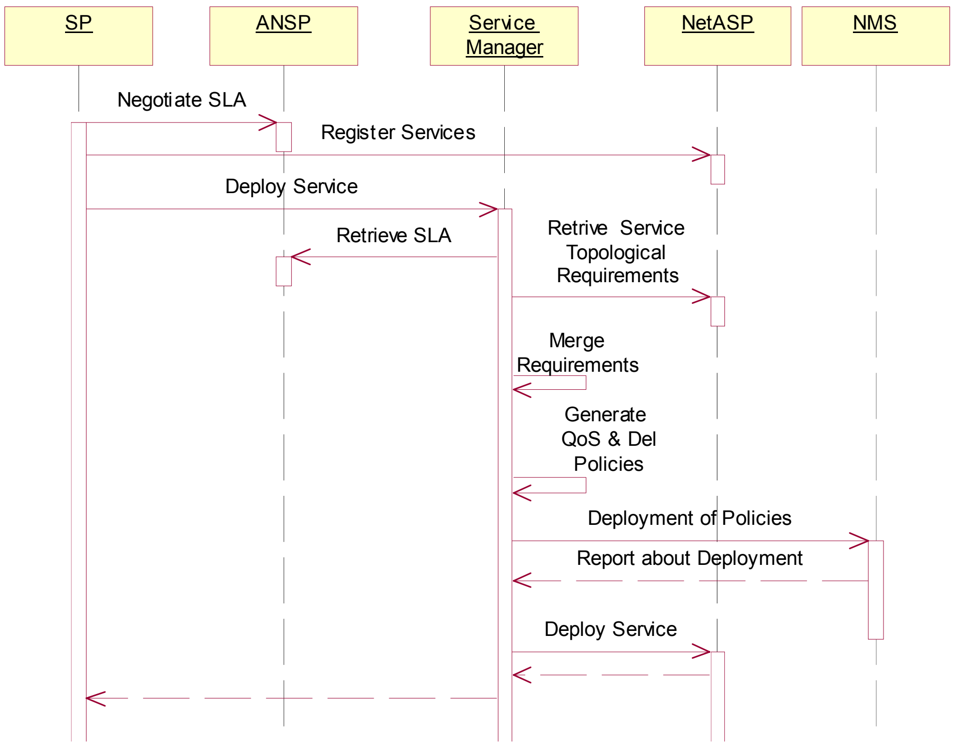


Figure 29. Setup of a VAN for Deploying a Particular Service

2.3.3.3 *Inter Domain Manager*

2.3.3.3.1 *Objective*

To implement end-to-end negotiation as a service across the Internet, it is inevitable that the traffic flow for the service must propagate across different domains. We define a domain as a collection of nodes by a single administrative entity where security and management policies are uniformly applied. The different administrative domains under discussion are owned by separate organisations.

The network management system must abstract the service request of an end-user within its domain to present it to the target domain. For this reason, we assume that both source and destination domains use the FAIN PBNM; and as such, will understand the service parameters for a particular request.

2.3.3.3.2 *Relation to previous work*

In the WebTV scenario, the role of this component is highlighted for reservation of computational and communicational resources for service deployment, particularly as requests occur between different administrative entities. We note that even in the case of this simple scenario, we are effectively propagating the video traffic across domain boundaries.

In our previous milestone (M5), we did not address how the administrator of Domain 1, *i.e.*, its ANSP, made sure that the administrator of Domain 2 would adequately provision the VE resource parameters on the active node in Domain 2 to allow the transcoder to run. Domain 2 is, in fact, an ISP to the group of customers, and nothing more.

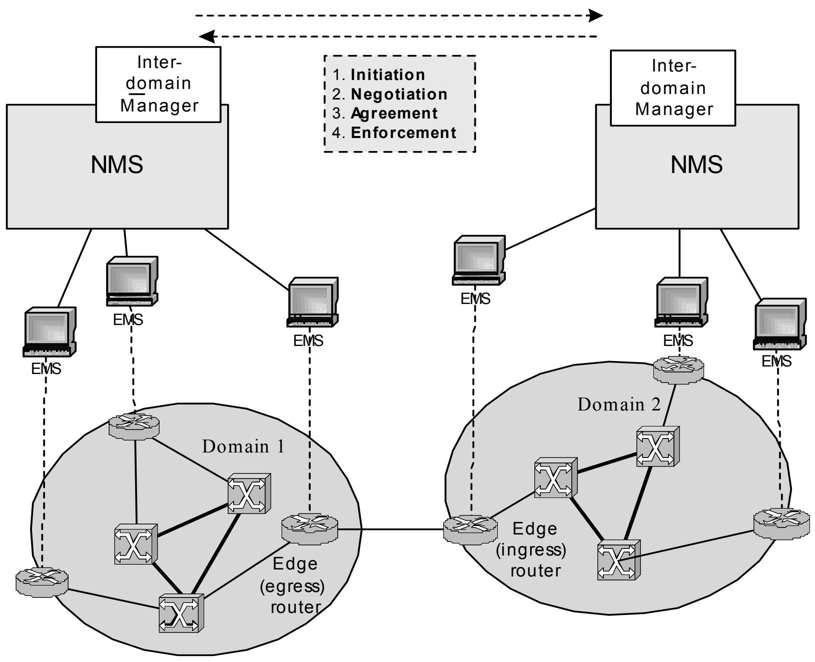


Figure 30. An abstract depiction of the inter-domain manager

2.3.3.3.3 Relation to enterprise model

We suggest two options for the SP to propagate its reservation sequence throughout the Internet domains in which it wants to set up a VE.

- An SP could separately negotiate with each ANSP the parameters for resource reservation per domain via the ANSP's network-level management system; or
- An SP could set out its requirements to a single ANSP, which would, in turn, negotiate the desired VE parameters with its neighbouring domains, in order to satisfy the 'least common denominator' for an agreement between their respective local policies

The end result for both options is a VAN established for the SP that encompasses multiple administrative domains, achieved by means of an inter-domain management property within the FAIN PBNM system.

As an example, when a WebTV client, say, at Domain 2 requests a service from the portal located in Domain 1, the view of separate domains is transparent to the client as well as the SP, which is the WebTV provider. An SP can request anything it wants, it is up to the ANSP to consider its capabilities, by further negotiating with other ANSPs.

The IDM comes in when ANSPs need to negotiate with each other. The Resource Manager will determine if this customer is in another domain. In IDM negotiation, if, for instance Domain 2 cannot fulfil Domain 1's requirement, it could offer a lower value. A successful negotiation will lead to a request to create a VAN from AN1 (ETH) to C1, this new branch will be added to the existent SP VAN.

As such, ANSPs need to derive contracts with other ANSPs to encompass the geographical spread of their targeted client base. For example, the administrator in Domain 1 needs to establish a relationship with Domain 2 in order to reach its customers.

2.3.3.3.4 IDM design and relationship to other components within the NMS

Each element management system (EMS) only manages a single active node. A network-level management (NMS) oversees these EMSs and the IDM is located within the NMS. We adopt the Java Remote Method Invocation (RMI) as the communication channel for the distributed system.

In our design we established a repository that maps a destination address to an IDM, *i.e.*, a 'many-to-one' relationship. As such, an ANSP must register a list of destination addresses within its domain on a repository that has a well-known address, so that this repository can provide a discovery mechanism for mapping destination addresses to their respective IDM.

Within an NMS, the IDM interfaces with two other key components, *i.e.*, the Resource Manager (RM) and the Service Manager (SM) (see §§ 2.3.3.4and 2.3.3.2).

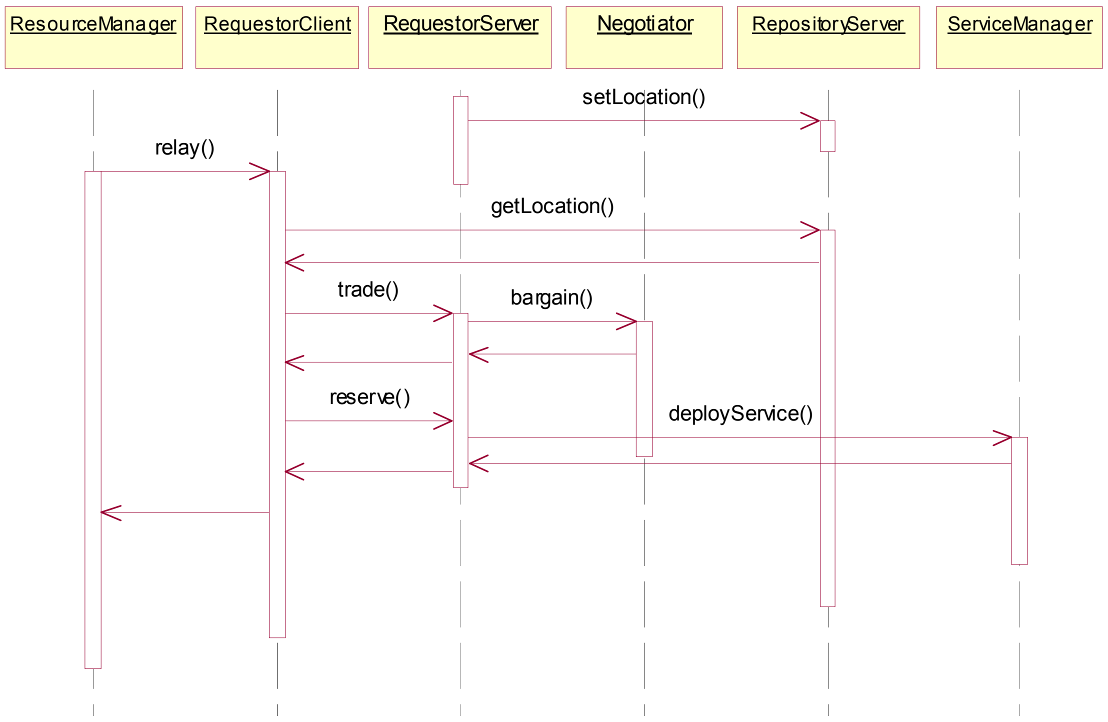


Figure 31. Sequence diagram for the Inter-domain Manager

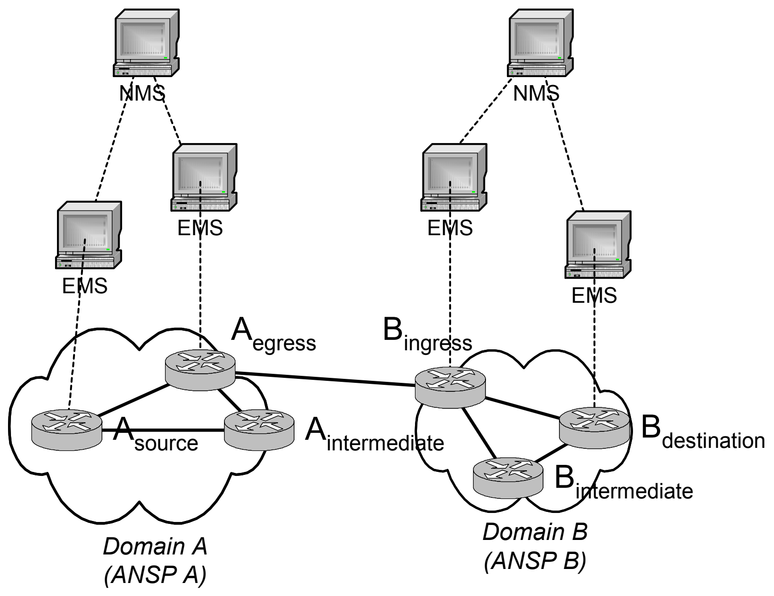


Figure 32. Simplest form of inter-domain interaction

The effect of considering an inter-domain view is as follows. The Resource Manager will invoke the `relay (Context, ServiceDescriptor)` method on the IDM in Domain A. Context represents the ingress IP and the destination IP of the reserved route. For ServiceDescriptor, it would be sufficient to input the service component name. The RM should have no problem extracting this information from the VANPath information that it has. The intra-domain process is stalled at this point.

At the other end, the IDM in Domain A will request the IDM in Domain B to extend the VAN in the neighbouring domain. The SM's `deployService (ServiceName, VANNode[], Credential)` method will be invoked by the IDM in Domain B. The VANNode array is a pair of ingress-destination IP address abstracted form the context information obtained from the RM.

This is the IDL description for the NMS interface is as follows:

```
   #ifndef _idm_IDL_
   #define _idm_IDL_
   #include <netasp.idl>
   #pragma prefix "ist_fain.org"

   module nms {
      module idm {
         typedef string IPAdd;
         typedef string QOSParameters;
         struct Context { //information extracted from network-level QoS
policies forming the context
            IPAdd ingress;
            IPAdd destination;
               };
               struct QoS {
                     QOSParameters bandwidth;
                     QOSParameters cpu;
                     QOSParameters memory;
               };
               // Provides method for inter-domain reservation
               interface Requestor {
                     exception InvalidRequest {};
                     void reserve((in
org::ist_fain::network::netasp::ServiceDescriptor descriptor,
                                  in Context c)) raises ((InvalidRequest));
                     String trade((in QoS q)) raises ((InvalidRequest)); //
IDM negotiates with peer.
               };

               /** Gets (from a repository) the correct IP address of the
NMS and port
                *  that the peer IDM is listening to. Repository maps an
active node to an NMS.
                */
               interface Repository {
                     exception MappingNotFound {};
                     IPAdd getLocation((in IPadd destination)) raises
((MappingNotFound));
                     void setLocation((in IPadd destination));
               };

         };
   };

   #endif // _idm_IDL_
```

2.3.3.4 Resource Manager

The Resource Manager (RM) is a component of the FAIN Network-Level Management Architecture, which maintains a global view of the connectivity and the availability of resources in one managed domain. The RM is used during the creation of a new virtual active network, for the establishment of an end-to-end path for the installation of an active service. During these processes, the RM has to communicate with both the QoS PDP and the ASP Network Manager.

First of all, the RM has to maintain the connectivity information of the active network. For this reason it stores information about all the nodes and the links of the system. For each node, the available properties are its public IP address, the private address, which the node may have inside the FAIN testbed, the types of Virtual Environments, which are supported by the particular node, the links attached to the node and finally the address of the Element Manager, which is responsible for this node. For each link, the available properties are the IP addresses of the start and end nodes, the total capacity of the link and the currently available bandwidth of the link.

Additionally, the RM stores a list of the Virtual Active Networks, which have been established in the system. Each VAN is associated with a unique identifier.

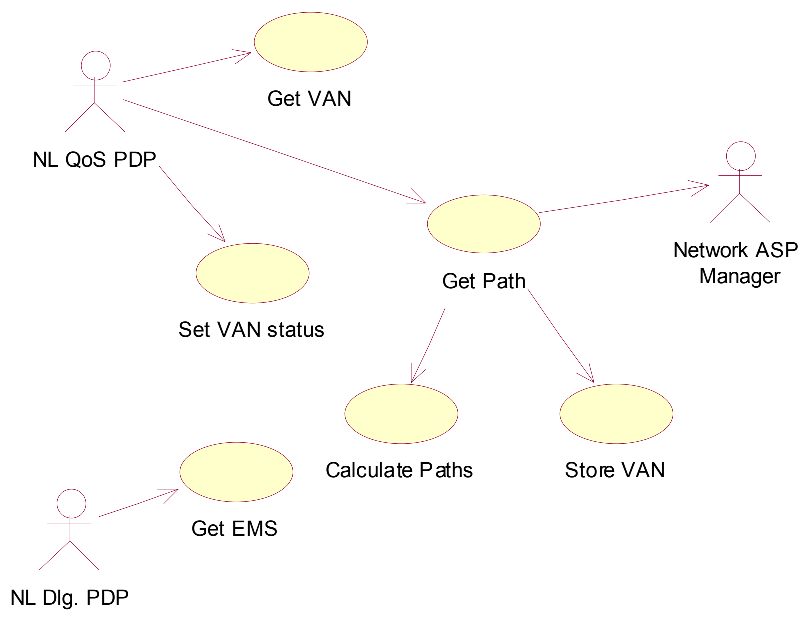


Figure 33. Resource Manager Use Cases

2.3.3.4.1 *Interface and operations*

The interface offered by the RM is displayed below:

```
interface ResourceManager {
   org::ist_fain::apbm::t_Parameter getPath(
                       in string ServiceName,
                       in string ActionMode,
                       in org::ist_fain::apbm::t_Parameter param);

   emsList getEMS(in string VANid);

   void vanStatus(in string VANid, in string status);

   org::ist_fain::network::VANPath getVANPath(in string VANid);
};
```

The operations included in the Resource Manager interface is as follows:

```
org::ist_fain::apbm::t_Parameter getPath(
                       in string ServiceName,
                       in string ActionMode,
                       in org::ist_fain::apbm::t_Parameter param);
```

This operation is called by the QoS PDP, for the creation or the extension of a VAN. The ServiceName input parameter is a string, which identifies the requested service. The ActionMode parameter specifies whether the current operation requests the creation of a new Virtual Active Network, or a modification to an existing one. The last input parameter, named param, contains a sequence of name-value pairs, describing the topological requirements for the Virtual Active Network.

The return parameter of the operation also contains a sequence of name-value pairs, specifying the created VAN.

```
emsList getEMS(in string VANid);
```

This operation is called by the Delegation PDP and returns a list containing the IP addresses of the EMS nodes, which are responsible for managing all the nodes that belong to the particular VAN, identified by its VANid.

```
void vanStatus(in string VANid, in string status);
```

This operation can set the status of a particular VAN. It is mainly used by the QoS PDP to remove associated information from the Resource Manager when the establishment of a VAN fails for some reason.

```
org::ist_fain::network::VANPath getVANPath(in string VANid);
```

This operation returns information about the nodes and links that belong to a particular VAN.

2.3.3.4.2 Functionality

The main role of the RM is to determine a suitable path for the installation of an end-to-end service. The procedure is triggered by the QoS PDP, which calls the getPath operation, providing all the resource and topological requirements of the service.

Looking at its internal database, the RM tries to find suitable paths in the network, which satisfy the requirements given by the QoS PDP. If the search is not limited by other constraints, e.g. choose shortest path, a set of different paths will result. All of these paths are candidates for the creation of the new VAN, which will accommodate the service.

The selected paths fulfil the resource and topological requirements of the service. However, a service can have additional requirements, which can be extracted from the service descriptor. The Resource Manager does not have access to such information, as this lies within the domain of the ASP, so the ASP Network Manager makes the final decision.

The Resource Manager sends the list of the valid paths, to the ASP Network Manager, via the calculateBestCandidates operation and gets as output the most suitable path. For each path, the Resource Manager also includes information about the properties of its nodes and links, based on which the ASP Network Manager will be able to perform its requirements matching and identify the most suitable path.

When the final path is returned by the ASP Network Manager, the Resource Manager stores the new VAN and updates all related information, to reflect the establishment of the new VAN. Finally the information about the new VAN is sent back to the QoS PDP, which will enforce the necessary actions for the establishment of the VAN. If for some reason the new VAN cannot be successfully setup, the QoS PDP can rollback the process by setting the status of this VAN to “FAILED”, which leads to the removal of all information associated with it.

The Resource Manager also supports extension of an existing VAN, e.g. in the case where a Service Provider has already obtained a VAN, but wishes to install additional service components to offer his service to new customers. VAN extension follows the same procedure as VAN creation, except that existing nodes are also taken into consideration.

2.3.3.5 Monitoring System

The monitoring system at the network-level is connected to the diverse notification channels at the element-level, as any other consumer, providing the event correlation capabilities required to obtain a precise picture of the active network status. The definition and design of appropriate filters avoids event-flooding and provides the necessary degree of scalability. The monitoring policies at the network-level define the composite events as a set of simple events produced in a specified order. From the monitoring policy, appropriate filters are generated by a composite event controller and sent to the EMS event channels. At the same time, appropriate components are configured in the NMS in order to detect the appearance of a succession of events coming from either the s*ame or different EMS stations.*

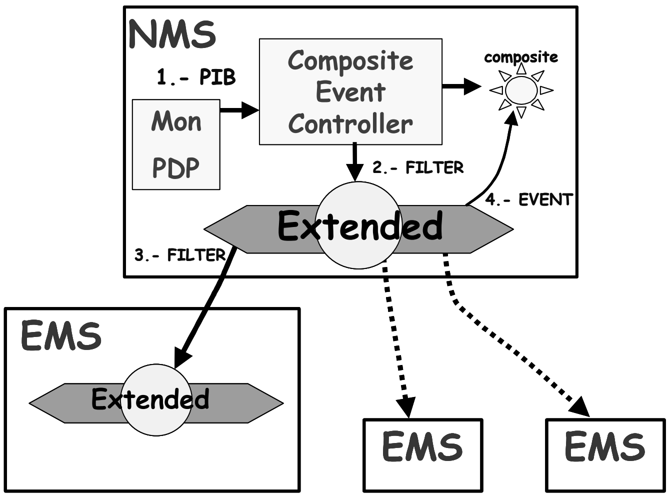


Figure 34. The Main NMS Monitoring System Components

The composite event controller generates composite event instances that become event consumers subscribed to receive the events it is composed of. The configuration of the composite event leads to the definition of an event matrix. Whenever an event is received by the composite event instance, it checks the order of appearance (relative to the other events) and stores it in the event matrix. When the event matrix is completed, a composite event is raised and appropriate alarms are generated in the network level. The NMS monitoring system may also act as a server that simply gathers all the events following a certain pattern or being associated to a given technology domain.

2.3.3.6 Quality of Service (QoS) PDP

The Quality of Service Policy Decision Point (QoS PDP) is a particularisation for QoS configuration of the core PDP. That is, policies processed by this PDP are oriented to the differentiation of certain flows, or groups of flows, with an enhanced quality of service. At the network level this PDP component plays a fundamental role within policy-based management architecture.

To develop all these capacities the QoS PDP needs to interchange information with monitoring system and resource manager components and make proper decisions based on policy conditions and network or node status. Besides, QoS PDP contains two types of components: the condition and action interpreters. They provide action and condition processing logic for those policy types, which are handled by this PDP. The QoS PDP can be dynamically extended by contacting the ASP system to accommodate additional interpreters capable of processing new actions and conditions conveyed by the policies. The more important interactions of the component are showed in Figure 35.

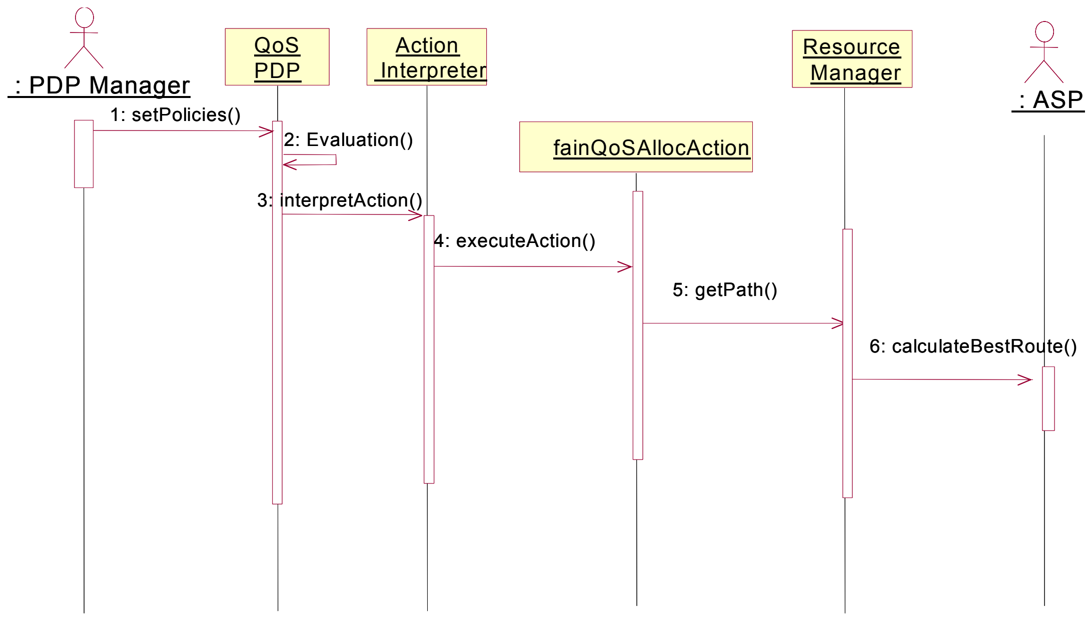


Figure 35. Policy process by QoS PDP

As a part of our management system design, we have split the VAN specification into three main branches: QoS Parameters, Computational QoS Parameters and Specific Service Requirements. When QoS PDP receives a policy, only information relative to QoS Parameters like bandwidth and priority are used, as previously mentioned this data are obtained from SLA between ANSP and SP. The corresponding information of the other two branches is obtained from resource manager through the monitoring system and ASP (details of this interaction are explained in RM sub-chapter).

The Action Interpreter contains the functionality to interact with the resource manager and monitoring system components in order to request from them the path of actives nodes that will be part of the VAN, as well as the information relating to computational QoS parameters and service requirements, in which case this request needs to have additional service information, mainly the name of the service, Virtual Network ID (VNID) and action mode (i.e. active, remove, modify, etc).

Once the resource manager sends back the chosen path, the QoS PDP adds this information to the rest of VAN requirements and integrates them in a structure that is forwarded to the QoS PEP in order to generate and distribute element level policies. The next section shows the information model interchange between RM and QoS PDP.

2.3.3.6.1 RM – QOS PDP interaction information model

We have two basic information structures, these are: "VAN Information" and "Topological Information" within RM - QoS PDP interface as shown in Figure 36. The first structure contains the information relating to VAN resource reservation while the second contains the information relating to VAN user requirements. Below we have referenced the complete description of each structure.

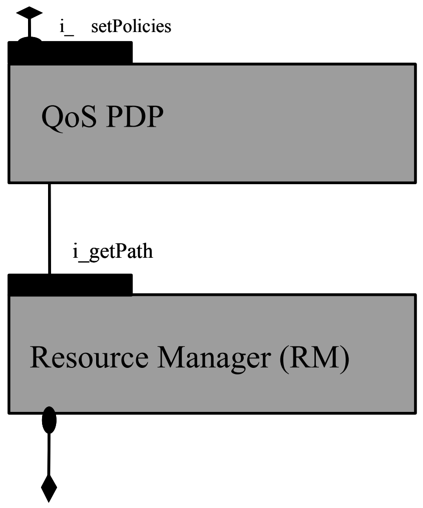


Figure 36. Architectural Model for QoS PDP – RM Interface

2.3.3.6.2 Topological Information

```
Name:     LinkConditions (This is the structure reference name)
Value:
   Name:  IngressLink
   Value:
         Name:  QoSClass
         Value: xxxxxxxx
         Name:  RequestBW
         Value: xxxxxxxx
   Name:  EgressLink
   Value:
         Name:  QoSClass
         Value: xxxxxxxx
         Name:  RequestBW
         Value: xxxxxxxx
   Name:  IngressIP
   Value: xxxxxxxx
Name:     EgressIP
   Value: xxxxxxxx
```

The structure above is almost fixed compared with those following, due the fact that for each service to deploy a VAN is necessary and this always has an Ingress and Egress IP, whose values for our case are IP network address and mask (i.e. 10.0.0.1/32). The links element of the structure contains, for example, information relating to the level of QoS and amount of bandwidth requested by the user , both of which are integer numbers.

In order to activate a VAN the resource manager sends back a structure with information about the target nodes, computational QoS parameters and service requirements. We show this structure below. In the first part of the structure there are two values, firstly the ActionMode corresponding to the kind of action that is requested from QoS PDP such as "activation" or "modify", secondly VNId refers to the identification of the VAN that is assigned by QoS PDP and is returned because is the case of VAN extension we need to know this value from RM.

In the second part we have designed a sub-structure that contains the information relating to Active Node (AN), as a special characteristic this sub-structure will be the same for all the ANs involved in the VAN, with just the internal values being different. The meaning of each value is as follows:

IPAddress: The network IP address with mask of the Active Node

EMS: The network IP address with mask of the EMS in charge of manage the AN

CPUPriority: An integer that indicates the range of policy importance or urgency

maxDiskSpace: The amount of hard disk space

maxMemory: The amount of memory

EEId: The kind of Execution Environment (EE) needed in the AN

As a general rule, if an AN is part of a VAN but no resources need to be reserved, then all the fields involved remain blank.

```
Name:     TargetNodes
Value:
Name:     ActionMode
Value: xxxxxxxx
   Name:  VNId
   Value: xxxxxxxx
   Name:  AN1
   Value:
          Name:  IPAddress
          Value: xxxxxxxx
Name:     EMS
          Value: xxxxxxxx
          Name:  CPUPriority
          Value: xxxxxxxx
          Name:  maxDiskSpace
          Value: xxxxxxxx
          Name:  maxMemory
          Value: xxxxxxxx
          Name:  EEId
          Value: xxxxxxxx (or empty, it depends on ASP information)
```

In the case of VAN Extension we have an additional structure, shown below. The main difference from the last one is the name of the ANs involved, as we include only those nodes that are new in the VAN, while the rest of the fields remain the same.

2.3.3.6.3 VAN Extension

```
Name:     TargetNodes
Value:
Name:     ActionMode
Value:   4  (Because 4 is the number for “modify” action)
Name:     VNId
   Value:
   Name:  Extended_AN1
   Value:
          Name:  IPAddress
          Value: xxxxxxxx
          Name:  EMS
          Value: xxxxxxxx
          Name:  CPUPriority
          Value: xxxxxxxx
          Name:  maxDiskSpace
          Value: xxxxxxxx
          Name:  maxMemory
          Value: xxxxxxxx
          Name:  EEId
          Value: xxxxxxxx (or empty, it’s depends of ASP information)
```

2.3.3.7 Quality of Service (QoS) PEP

The Quality of Service Policy Enforcement Point (QoS PEP) component at the network-level has, from the conceptual point of view, the same functionality as that at the element-level, except that signalling support is not provided at the network level, for reasons given in the common components section. However, the actual processes needed to realise the functionality (i.e. translation of policy decisions into commands that are understandable by the target) vary significantly since these processes at the network-level are policy translations from network to element-level policies. This fact is reflected in the use case diagram shown in Figure 37.

The particularities of the network level QoS PEP in relation to the previously described element-level QoS PEP are:

- Support for integration of computational service requirements, together with the topology and quality of service requirements, concerning so that element-level policies can be generated.
- Distribution of policies to different EMSs in different systems from the target active node. In the past it was necessary that the target node and EMS resided in the same machine in order for NL QoS PEP to distribute policies to the EMSs.
- Previously the distribution of policies did not include any status control. One of the most important changes in our design is policy status support, by which, for instance, the QoS PEP receives several status reports from each of the element-level management stations that received a policy to process.
- One important addition to this component is the ability to process several policies at the same time, in order to enhance the performance of the policy distribution to EMSs. With this facility our management system is able to deploy many VANs within the ANSP domain in parallel.

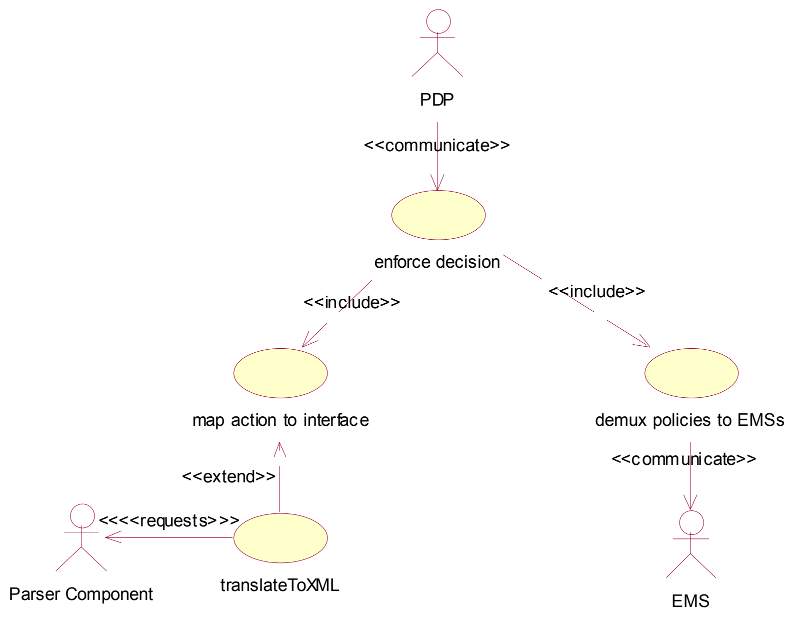


Figure 37. NL QoS PEP Use Cases

2.3.3.8 Delegation of Access Rights PDP

The Access Rights Delegation PDP is used to process Delegation of Access Rights policies. These policies specify to what extent an actor is permitted to access network resources through the control interfaces provided. In this way, by controlling access to resources, the operations on them are also controlled, eventually restricting the capabilities of services that are deployed. In FAIN, the Delegation PDP is used by the ANSP to determine what operations the SP's services are allowed to carry out on those network resources assigned to the SP as part of his virtual network creation. Delegation of access rights policies involves the (re)configuration of the security components of those nodes that form the topology of the SP virtual network. Every request to use a particular interface is checked by the security component. Access is only granted to authorised entities.

At the Element Level, the Delegation PDP extracts the necessary security information from the Delegation policy and then passes them to the Delegation PEP for enforcement.

The scope of the Network-level Delegation of Access Rights PDP is the same as in the Element-level Delegation PDP (section 2.4.3.3). The only difference in the network-level, is that the functionality of the Delegation PDP is more limited and after syntax checking the NL Delegation policy, it sends it to the NL Delegation PEP for further processing. The use case diagram is shown below:

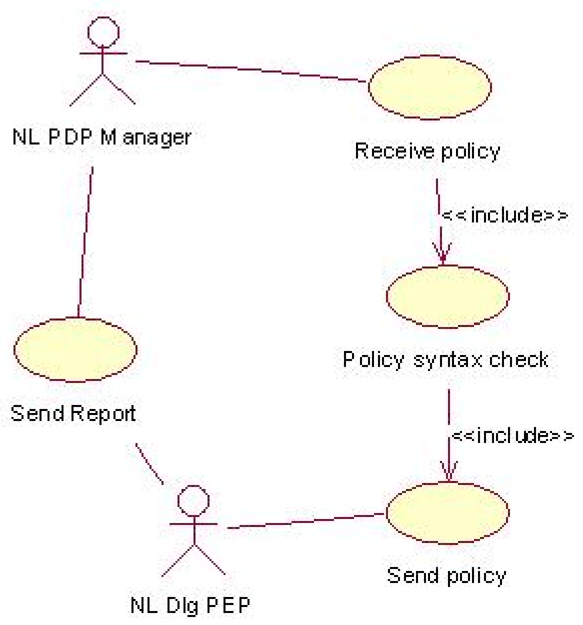


Figure 38. Use case diagram of the NL Access Rights Delegation PDP

The sequence diagram that shows the main interactions of the NL Delegation PDP is the following:

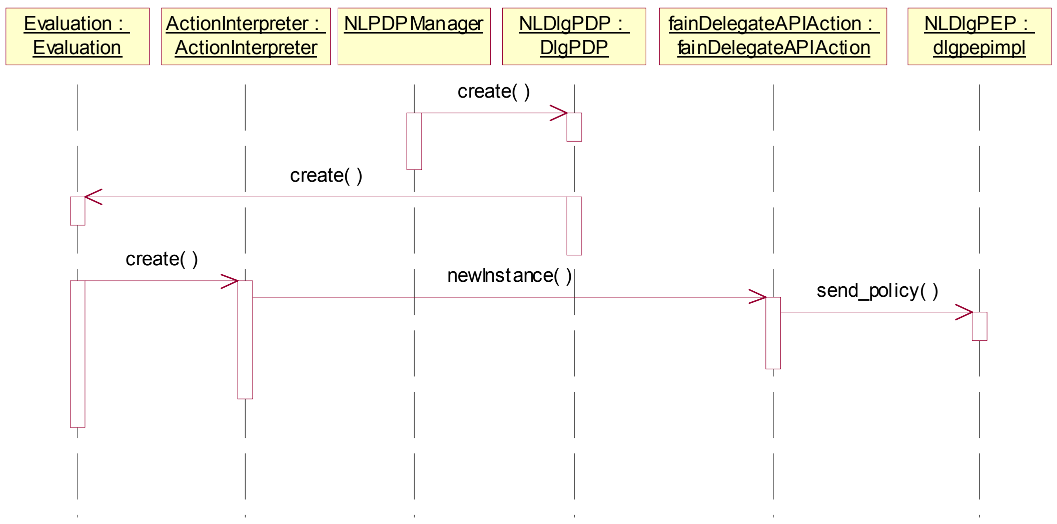


Figure 39. Sequence diagram of the NL Access Rights Delegation PDP

2.3.3.9 Delegation of Access Rights PEP

The NL Delegation PEP, after receiving the Delegation policy from the NL Delegation PDP, extracts certain parameters from it, and creates the Element Level delegation policies. It contacts the Resource Manager to retrieve the Ids of the nodes on which the policies should be enforced, and then sends the EL Delegation policies to the EL ANSP Proxy. Two types of Policy are created: the first type describes the Access Rights that will be eventually be assigned to a component in a node, and is called the Access Rights policy; the second type describes the components that will be instantiated in a new domain, and. is called the Delegation of Management Functionality policy.

The *Deleml1gen* class shown in the diagrams below will generate the Access Rights policy whereas the *DelMgmteml1gen* class will generate the Delegation of Management Functionality policy. The parameters that will be retrieved from this type of policy will eventually be delivered to the PDP manager (via the ANSProxy) that will perform the instantiation of the components in the new domain. The use case diagram is depicted below:

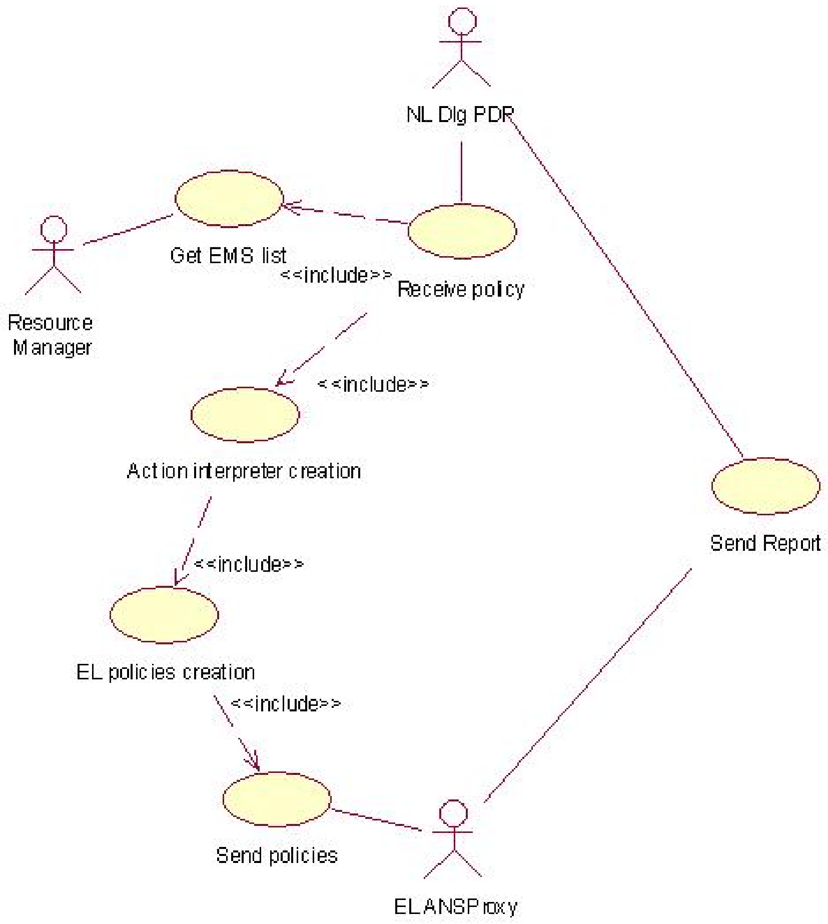


Figure 40. Use case diagram of the NL Access Rights Delegation PEP

The sequence diagram that shows the main interactions of the NL Delegation of Access Rights PEP is the following:

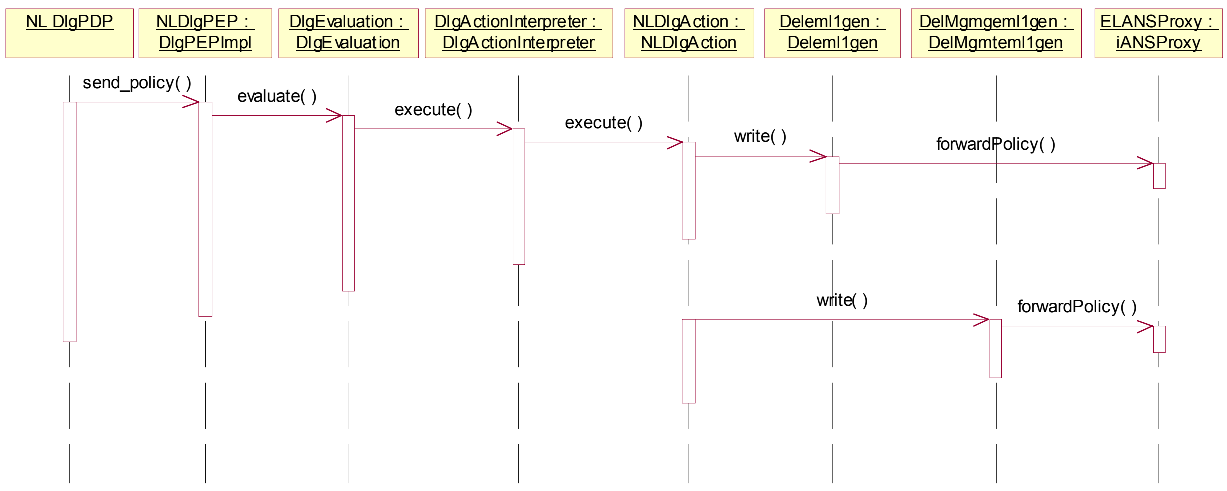


Figure 41. Sequence diagram of the NL Access Rights Delegation PEP

A simplified class diagram is shown below:

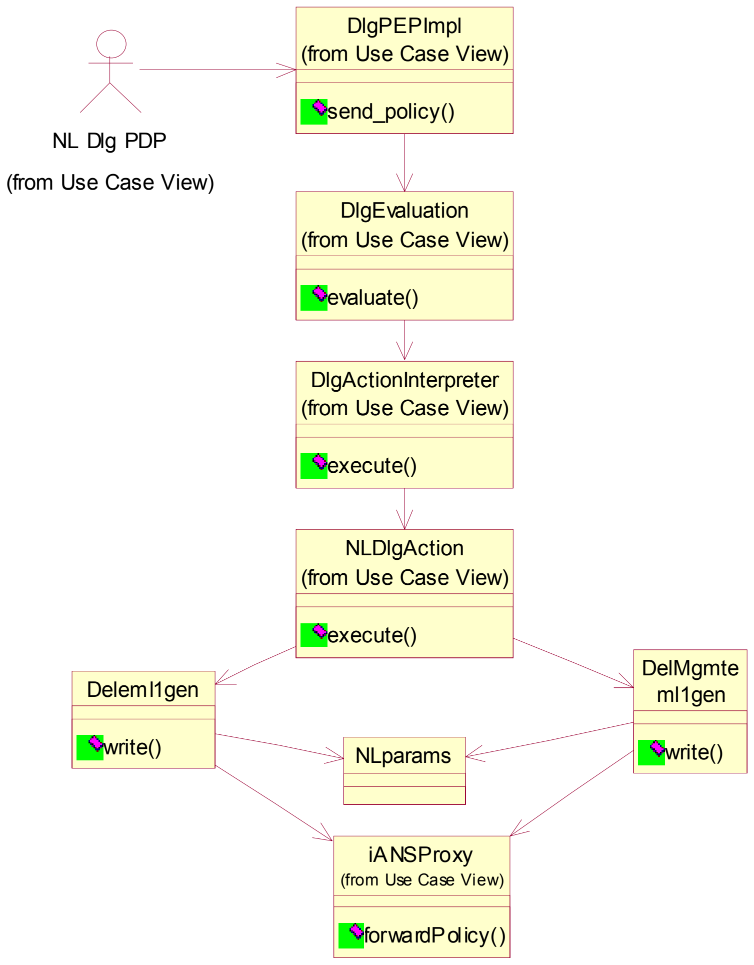


Figure 42. Class diagram of the NL Access Rights Delegation PEP

2.4. Element-level Management System (EMS)

Figure 31 shows the main components of the EMS. We can also see here the common components described in the previous section. In the following sub-section we will explain first the main interfaces used to communicate with components outside the EMS. In the rest of the sub-section we will briefly describe those components that are particular to the element management system.

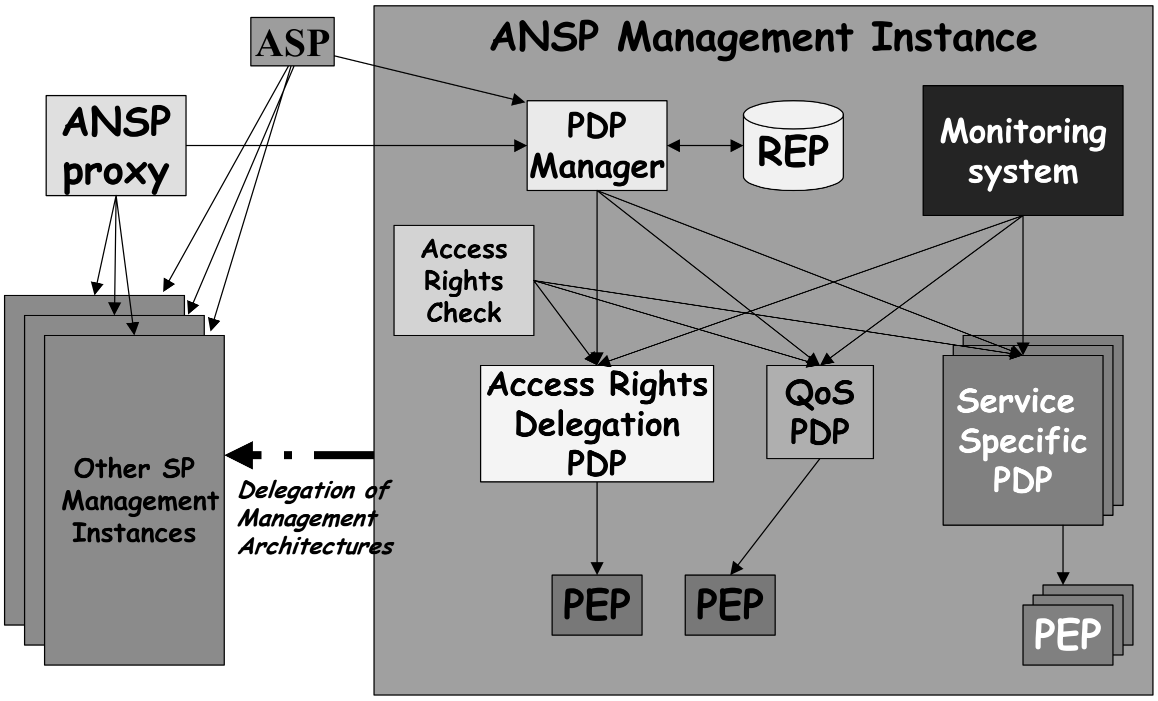


Figure 43. Architectural Model for EMS

2.4.1 Use cases

This section introduces the main use cases, which are more closely related with the element level than with the network level. The use cases to be covered are signalling, policy within active packet and the fault-triggered management reconfiguration, as highlighted in Figure 44.

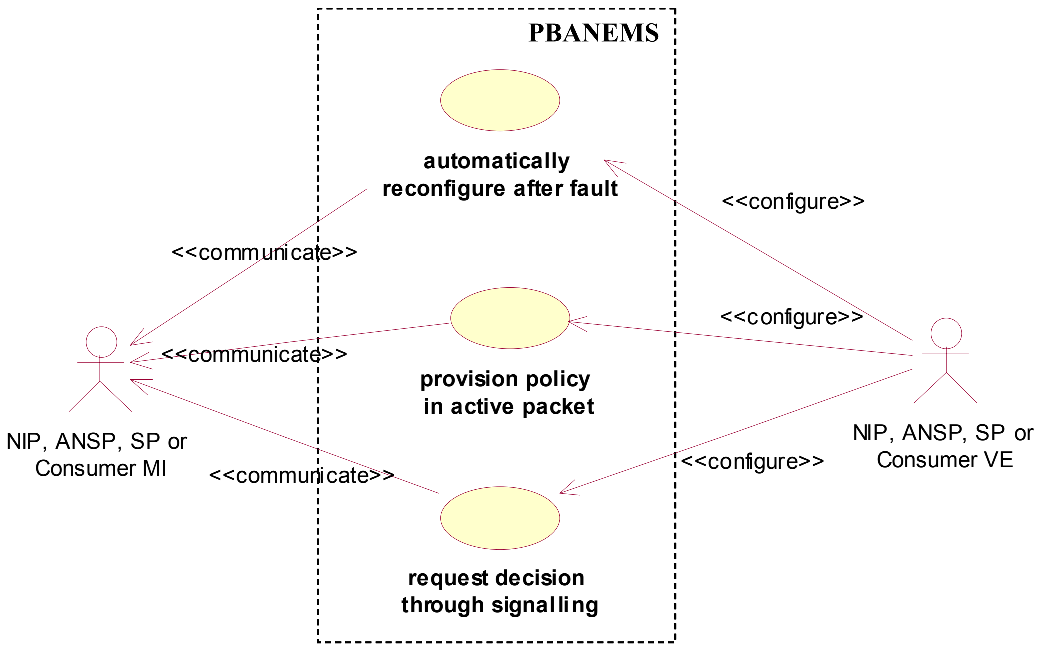

PBANEMS
automatically reconfigure after fault
provision policy in active packet
request decision through signalling
<<communicate>>
<<communicate>>
<<communicate>>
<<configure>>
<<configure>>
<<configure>>
NIP, ANSP, SP or Consumer MI
NIP, ANSP, SP or Consumer VE

Figure 44. Use Case Diagram for EMS

Before proceeding with the actual description of the use cases, please refer to the page xxx for the list of acronyms used. This actor appears in the element level use cases because most of the changes occurring at the element level are notified to the network level. Thus, the network level always has a general view of the network resources and can act accordingly.

2.4.1.1 Automatic Reconfigure after fault

This use case is responsible for readapting the active node and network configuration when a fault occurs.

The management framework, upon receiving the alarm of a fault occurrence, will determine the policies that should be applied to correct the faulty situation. In that way the system is autonomous, distributed and resolves problems and faults quickly.

The policy logic functionality in this use case is mostly the same as in the policy provisioning use case with the exception that there is no new policy introduced, thus there is no need to store any new policies in the policy repository.

As described above, the management instance will try to solve the problem, and if it succeeds, then the configuration changes should be notified to the appropriate network level management instance. However, it might be the case that for several reasons, the problem can not be solved at the element level, then the element management instance will send an alarm notification to the network level management instance to allow it to react accordingly.

2.4.1.2 Provisioning Policy contained in Active Packet

The provisioning policy contained in Active Packet use case is the same as the provisioning policy use case. There are new functionalities specifically added to the element management level. A policy is included within an active packet and forwarded to the nearest management station each time it arrives at a targeted active node. Since the nearest management stations to active nodes are the element management stations, it is, in practice, applied only at the element level. Theoretically the use case is applicable at both levels of the framework. The policy is stored at the element level policy repository and a notification is send to the corresponding network level management instance. If the policy had been sent by the network level this notification serves as a confirmation. Otherwise it informs that a new policy, with its main properties coming from the virtual environment, has been introduced at the element level management instance.

2.4.1.3 Request Decision through Signalling

The signalling use case specific to the element level management includes new functions:

- **Demultiplex decisions to PEP:** At the element level the PEPs are located within the active node, particularly within the VE owned by the same actor as the management instance where the decision is made. Therefore, since there might be the case that a single element management system's station manages several active network nodes, we will have a one-to-many relationship between decision points and enforcement points. Hence, the component that realises the 'make decision' function at the element level should be extended with the function necessary to find the appropriate enforcement point where this decision should be forwarded.
- **Dynamic conflict checking functionality:** There is a need for checking possible conflicts between different policies at the precise moment where the policy should be enforced. This functionality is realised in part when the decision is made and in part when the decision has to be enforced. The approach in FAIN is to avoid dynamic conflicts as much as possible, hence making a clear and efficient allocation of resources whilst keeping different allocations completely isolated from each other. Further dynamic conflicts will be dealt with as follows. When a policy has to be deployed, it will be checked for conflicts, within the PDP, against other policies in that PDP. If any conflict is detected, it will be resolved with policy priorities. If no conflicts exist in this first step, the element management system will maintain its normal process and will try to enforce the policy in the node. In this case, the enforcement point may find that there are insufficient resources, that is, it detects a dynamic conflict. As such, the element management system will only enforce that request if it comes from the owner of the infrastructure, usually the NIP. If enforced, the node should notify the responsible entities of the removed resources so as to allow them to react accordingly.
- **Notify Configuration Changes to NL MI:** When resources of others actors are released due to dynamic policy conflict, as introduced above, this function would be responsible for notifying the network level, which will, in turn, forward it to the relevant actors.

2.4.2 Application Programming Interfaces (API)

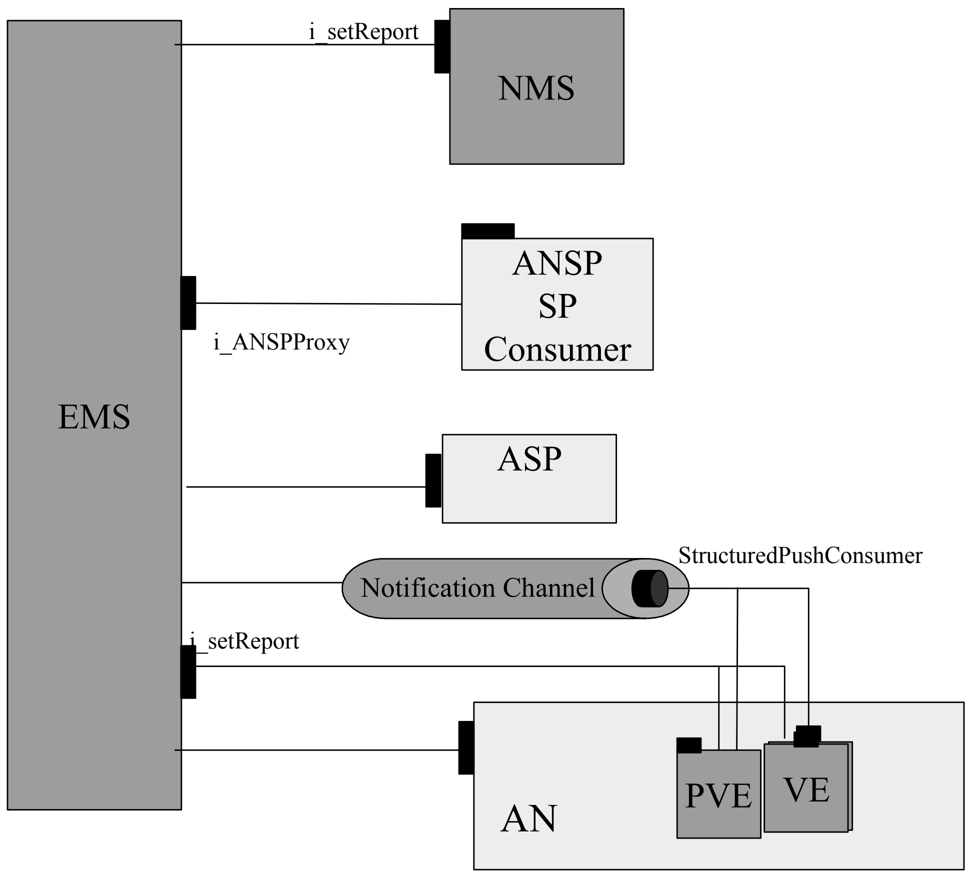


Figure 45. Interfaces Offered by EMS

This section describes the main EMS interfaces accessed by ANSP, SP, Consumer and other FAIN systems in order to develop management tasks over FAIN Active Nodes.

i_ANSProxy: This interface is offered by the ANSPProxy Component and is the initial access point for ANSP, SP, Consumer and NMS, as it receives a request in form of a policy. Once it is received a set of events are triggered in order to start the logic processing of the request.

This is the IDL description for the interface:

```
/** AnspProxy  Interface.*/
        interface iANSProxy {
          oneway void forwardPolicy((in string policy));
```

Where:

Argument Name	Argument Type	Description
policy	String	String containing the incoming policy. The policy is defined following the PCIM Information Model
Returns	Nothing. It is an asynchronous call. A one-way CORBA communication type.	

Exceptions	

i_setReport: This interface is offered by all the PEPs/PDPs and is used for the PEPs running as an active service inside the Privileged Virtual Environment (PVE) or VE (which are located inside AN) to send a report about the actual policy enforcement status.

This is the IDL description for the interface:

```
   /** i_setReport interface.*/
           interface  i_setReport {
             oneway void setReport((in t_Report  report));
   };
```

Where:

Argument Name	**Argument Type**	**Description**

report	t_Report	Structure that contains attributes to show the actual policy status enforcement. IDL type defined as follows:
		`enum t_policyStatus { INPROGRESS, DONE, FAILED };`
		`struct t_Report {`
		`string owner;`
		`string pdpName;`
		`string policyRef;`
		`t_policyStatus status;`
		`string details;`
		`};`

Returns	Nothing. It is an asynchronous call. A one-way CORBA communication type.	
Exceptions		

StructuredPushConsumer: provides the access point to the notification channel. Whenever the probes running in the VEs or the privilege VE detect an event, they forward it to the notification channel using this interface. The notification channel distributes them to the interested PDPs.

This is the IDL description for the interface:

```
   /** StructuredPushConsumer interface */
           interface  StructuredPushConsumer : NotifyPublish {
             void push structured event( in CosNotification::StructuredEvent
notification )
   raises (CosEventComm::Disconnected);
           };
```

Where:

Argument Name	• **Argument Type**	• **Description**
notification	StructuredEvent	Enable to map a wide variety of event formats to a common structure. Each structured event consists of an event header (with a fixed part and optional header fields), the filterable event body which enables applying filtering on the events and the

		remainder of the body which is container for transparently submit information to the PDPs.
• **Returns**	void	
• **Exceptions**	Disconnected: when the PDP (consumer) is no longer connected to the notification channel.	

2.4.3 EMS Components

2.4.3.1 Quality of Service (QoS) PDP

This PDP is responsible for processing Quality of Service (QoS) Policies. By QoS policies we mean those policies that allow differentiating a particular flow or an aggregated flow with an enhanced quality of service. Currently the only way to differentiate among flows is to assigning them to a different VE and assigning for each VE a particular resource profile. This resource profile consists of a set of resources available at node level.

In FAIN we use QoS policies to trigger the creation of a VE and setting up a resource profile. The resource profile at element-level is abstract enough to be able to give support to other implementations. For this reason the task of mapping these parameters to particular parameters understandable within the active node is performed by the PEP.

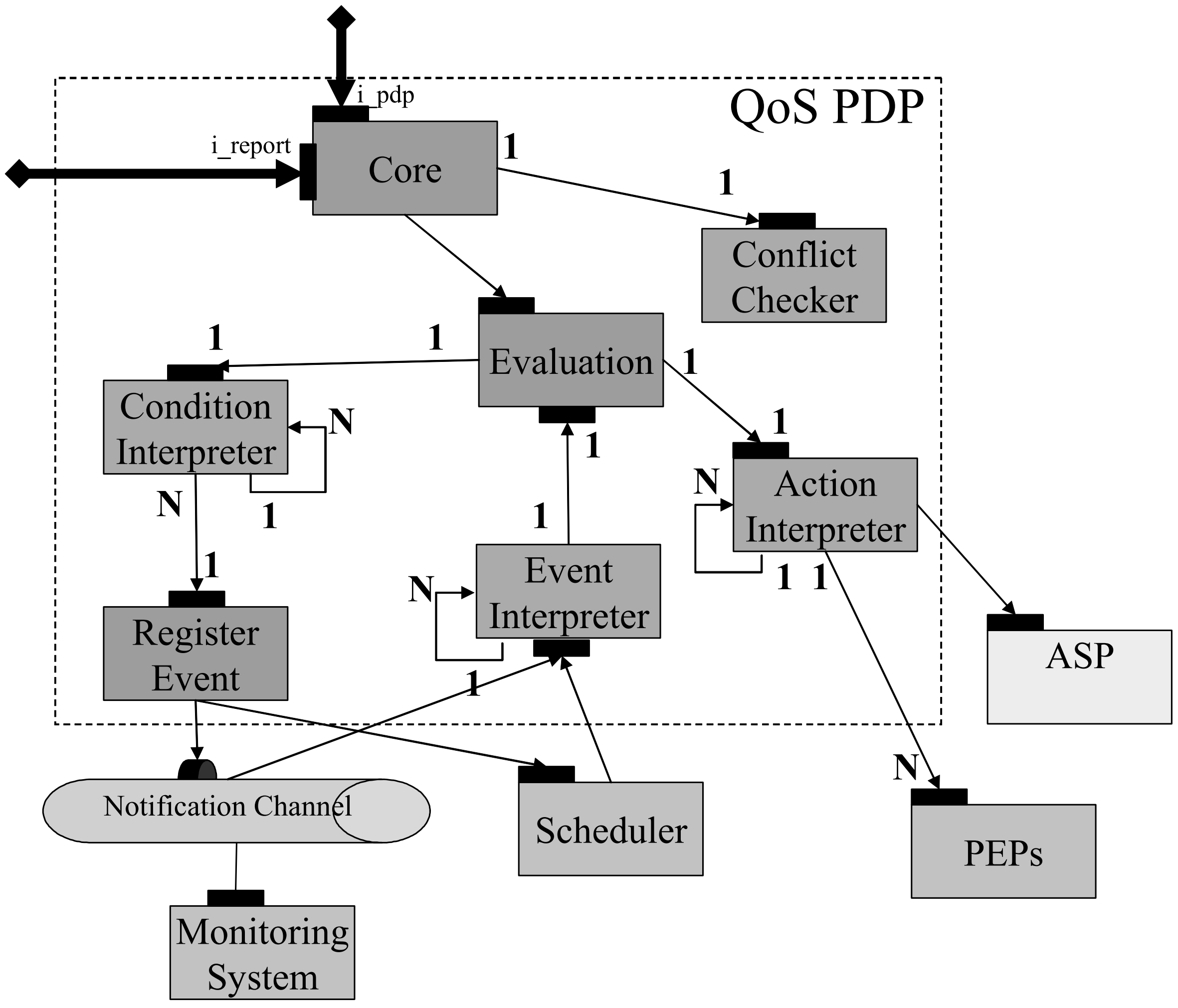


Figure 46. Architectural Computational Model for QoS PDP.

The Figure 46 illustrates the architectural components that contain a QoS PDP. It is identical to the Core PDP except that it includes more than just the generic Event/Condition/Action Interpreter. The QoS PDP was designed by extending the core PDP component and implementing all the condition, action, event interpreters required.

2.4.3.2 Quality of Service (QoS) PEP

The PEP (Policy Enforcement Point) component of the PDP is a very important part of the policy-based management architecture. Its main functionality is enforcement of decisions in the policy target (i.e. the active node). It supports two ways of working: provisioning (the interactions are initiated by the PDP with a decision) and signalling (the interactions are started by a decision request coming from the node interface.

The QoS PEP is responsible for the actual enforcement of the action enclosed into the QoS element-level policies. It is responsible for mapping the medium level abstract parameters into particular commands that are understandable within the active node.

In the particular case of the QoS element-level policies that create a VE, the PEP maps the medium level attributes into a resource profile. A resource profile is used by the Virtual Environment Manager (VEM) to create any referenced resource; currently the resources available are as follows:

Channel Resource: If it is specified, a channel resource will be created and all the component instances running inside the execution environment will be able to connect to it, and thereby receive and send packets from and to the network. The channel resource can be allocated a specific percentage of the bandwidth allocated to the VE.

Traffic Resource: If it is specified, a traffic controller will be created and the component instances running inside the EE will use it to control particular packet flows. Depending on the type of traffic controller used it will offer methods for setting up a guaranteed bandwidth or a specific packet scheduling.

DiffServ Resource: This is a specialised DiffServ traffic controller that allows the component instances running inside the EE to prioritise particular packet flows in preference over others.

Execution Environments: As explained in [D7], since the component instances must run inside an EE, at least one EE must exist, attached to a virtual environment. The FAIN active node has support for:

o JavaExecution Environment Resource: That's the more common EE. It provides a runtime support for service components implemented in JAVA together with support for inter-component communication based on CORBA.

o PromethOS Environment Resource: That is a High Performance EE and it is use always that the component instances have been implemented as PromethOS plugins.

o Snap Environment Resource: The SNAP execution environment features the execution of active packets and uses SNMP for communication with other component instances.

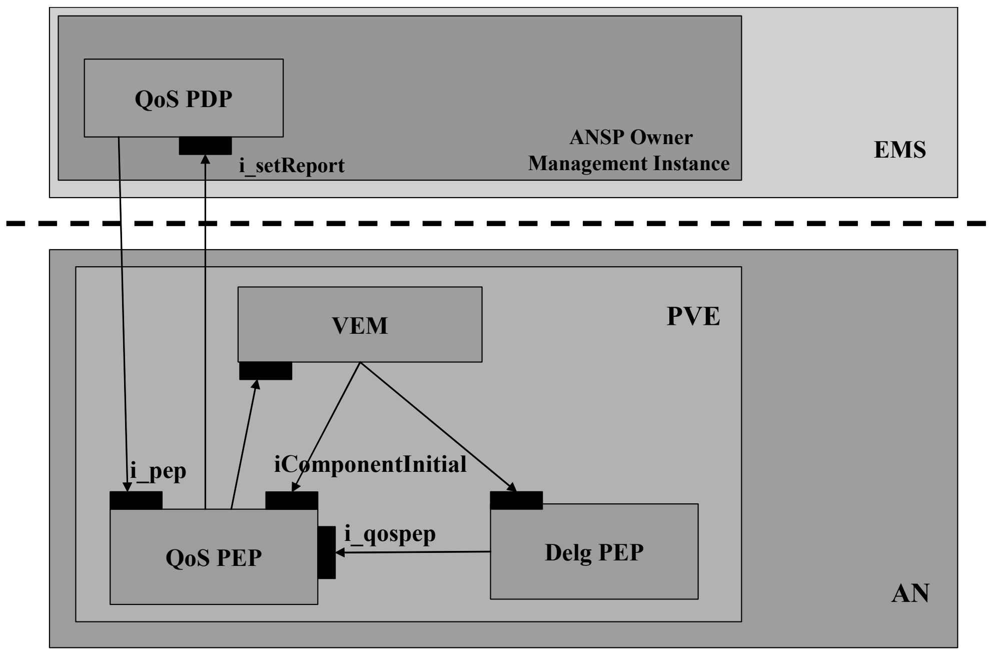


Figure 47. Figure 2: API offered by QoS PEP

Figure 2 illustrates the API offered by the QoS PEP and how it is accessed by its peer components. The black rectangles represent interfaces supported by the PEP component. The QoS PEP has been implemented as an Active Service Component in order to be able to execute inside a JAVA Execution Environment running in a VE.When the QoS PEP is owned by the ANSP it is deployed and executed inside the Privileged VE.

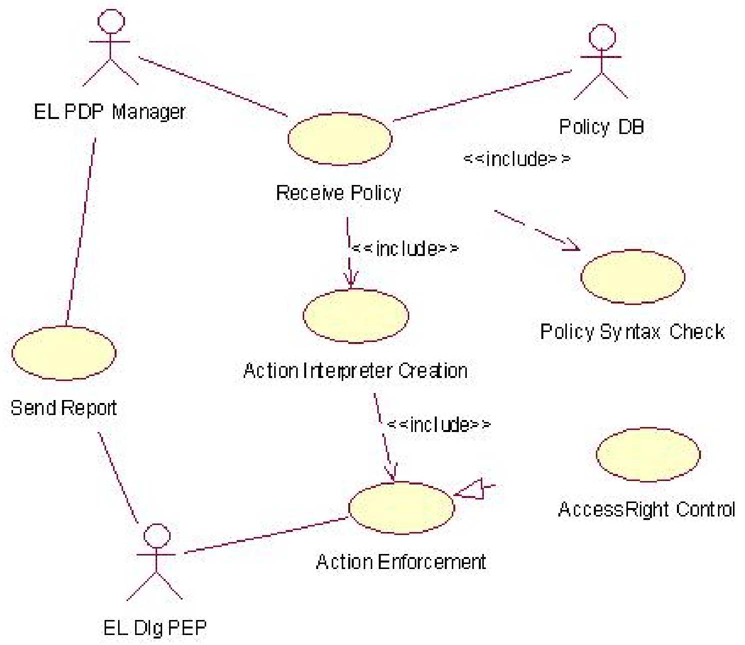
The QoS PEP provides the following interfaces:

- “i_pep”, is used by the QoS PDP to forward the decision taken to QoS PEP. (i.e. when a QoS Policy for creating a VE is received, PDP sends the appropriate command and arguments that allow to the PEP to create it).
- “i_qospep”, is mainly used by the Delegation of Access Rights PEP (i.e. when a VE has been created and a resource profile assigned to it, the VE still needs to be activated, and this activation is done by the Delegation of Access Rights PEP. So the Delegation of Access Rights PEP will contact with QoS PEP to retrieve the appropriated VEID associated with the VE created.
- “i_ComponentInitial”, as stated before the QoS PEP runs as an active service inside the Privileged Virtual Environment (PVE). So this interface is used by the Virtual Environment Manager (VEM) to retrieve all ports implemented by the component. In particular the other interface described above can be retrieved through this interface.

2.4.3.3 Delegation of Access Rights PDP

The Access Rights Delegation PDP is used to process Delegation of Access Rights policies. These policies specify to what extent an actor is permitted to access network resources through the control interfaces provided. In this way, by controlling access to resources, the operations on them are also

controlled, eventually restricting the capabilities of services that are deployed. At the Element Level, the Delegation PDP extracts the necessary security information from the Delegation policy and then passes the security information to the Delegation PEP for enforcement.



Use case diagram of the EL Access Rights Delegation PDP depicts the access rights control mechanism managed by the Delegation of Access Rights PDP. The PDP interacts with the Delegation of Access Rights PEP which consequently contacts the Security component of each of the active nodes. In this figure, VE1 and MI-1 are assigned to the SP-1 whereas VE-1 and MI-2 are assigned to the SP-2. The ANSP-MI and prVE are used by the ANSP. Access to the components of the active nodes are restricted to both users (SP1-1 and SP-2) by the access rights control that is initially defined in the delegation of access rights policies.

For example, SP-1 may have access rights attributes “ReadWrite”, “Read”, “ReadWrite” to Component-1, Component-2 and Component-3 respectively. This means that SP-1 can configure only Component-1 and Component-3 whereas it can only read information from Component-2.

With similar mechanisms we could implement a business scenario that involves a billing service provider who could be delegated to perform billing for other service providers. In such a scenario, SP-1 may outsource the billing task to SP-2, who is expected to collect usage information form each component in order to produce a bill for SP-1. SP-2 should be given access to all those components that are of interest to him.

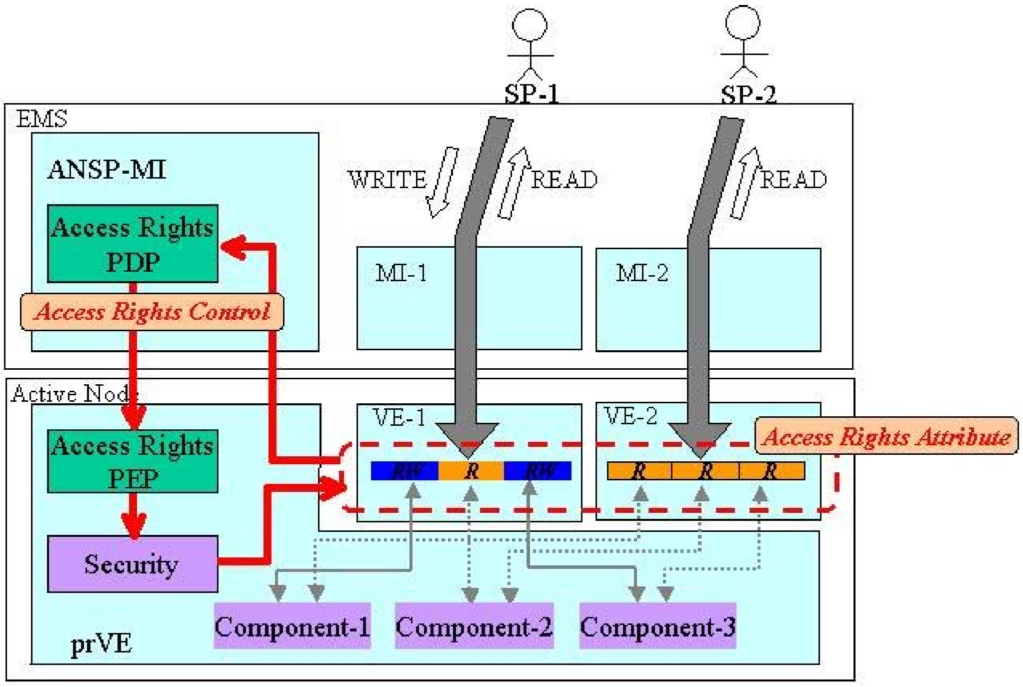


Figure 48. The Access Rights Control

The delegation of Access Rights PDP inherits features from the core PDP, such as the way that the Policy Action Interpreters are instantiated.

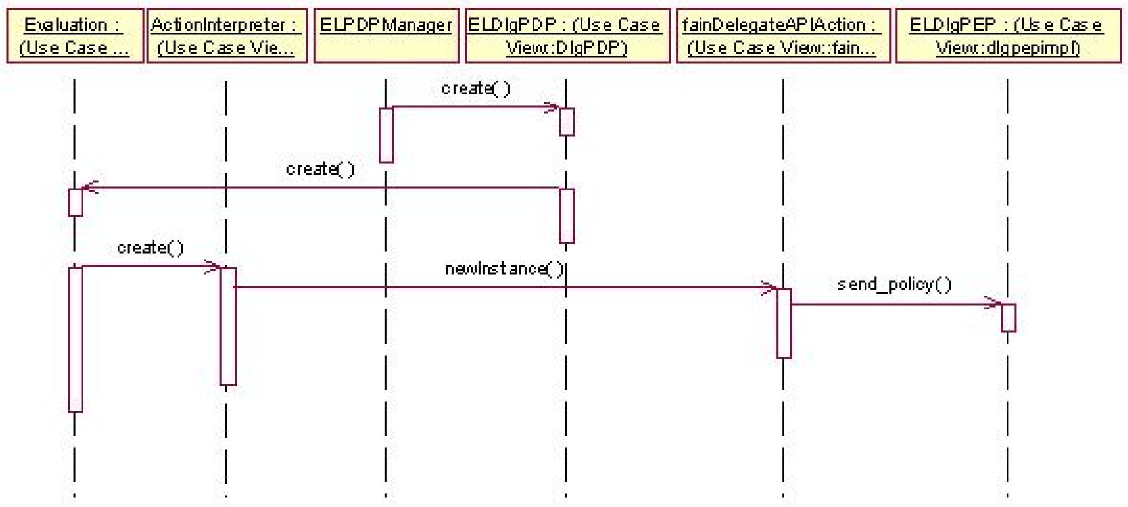

Evaluation : (Use Case ...
ActionInterpreter : (Use Case Vie...
ELPDPManager
ELDlgPDP : (Use Case View::DlgPDP)
fainDelegateAPIAction : (Use Case View::fain...
ELDlgPEP : (Use Case View::dlgpepimpl)
create( )
create( )
create()
newInstance( )
send_policy()

Figure 49. Sequence diagram of the EL Access Rights Delegation PDP

The sequence diagram showing the interactions that involve the EL Delegation PDP is shown in Figure 49. For completeness, we have included the creation mechanism of the Policy Action Interpreter and the fainDelegateAPIAction class, which is responsible for sending the Delegation of Access Rights parameters to the Delegation of Access Rights PEP:

2.4.3.4 Delegation of Access Rights PEP

The Delegation of Access Rights PEP at the Element Level runs as a service inside the Active Node (AN) and enforces the Delegation of Access policies to the AN. Specifically, it receives the security-related parameters from the Element Level Delegation PDP and passes them to the node management system so that they are enforced by the security framework within the active node. In order to identify the correct VE, the Delegation PEP retrieves a reference to the proper VE from the QoS PEP, which helped to create that VE. Its use case diagram is shown below:

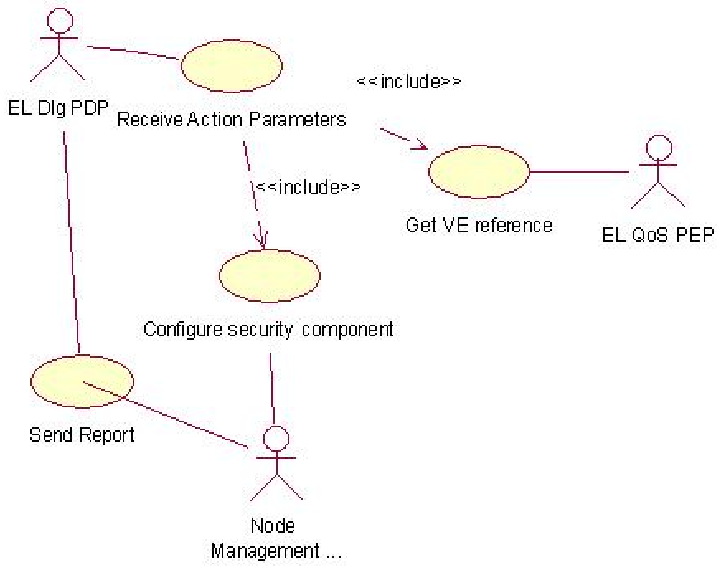


Figure 50. Use case diagram of the EL Access Rights Delegation PEP

The sequence diagram that shows the main interactions of the EL Access Rights Delegation PEP is the following:

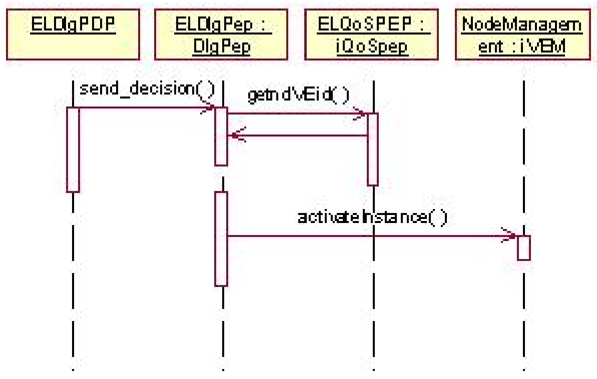


Figure 51. Sequence diagram of the EL Access Rights Delegation PEP

2.4.3.5 Monitoring System

The whole monitoring infrastructure is progressively created from the bootstrap. Figure 52 depicts the initial components existing in each layer: the monitoring PDP and policy repository in the control layer; the extended notification channel in the distribution layer and the master sensor registry in the acquisition layer. The rest of components are uploaded on demand, therefore decreasing the monitoring system overhead.

After this initial phase, the master sensor registry listens for any PDP subscription filter addressed to the EMS in which it is running. Such filters, originated by the PDPs during their event subscription processes, are delivered through the extended event channel to the master registry. The master sensor registry analyses them, extracting the configuration parameters they contain. It then examines whether there is already any sensor suitable for performing the requested monitoring operation on the considered target. If no sensor is available, it contacts the monitoring PDP asking for a configuration decision.

The monitoring PDP navigates the policy repository, gathering the policies applicable to the considered target and maps the policy actions into an appropriate and predefined monitoring PIB (Policy Information Base).

The monitoring PDP delivers the IDL-defined PIB, through the COPS interface [1], to the master sensor registry. In this case, the master sensor registry acts as the ‘PEP’ contact point. The master sensor registry will use the PIB and filtering information altogether for instantiating and deploying the sensors. Two possibilities are to be considered: in the local deployment, the sensor is installed in the EMS station itself, remotely accessing the monitored targets. Remote deployment requires accessing the ASP for installing the sensor in the corresponding active node.

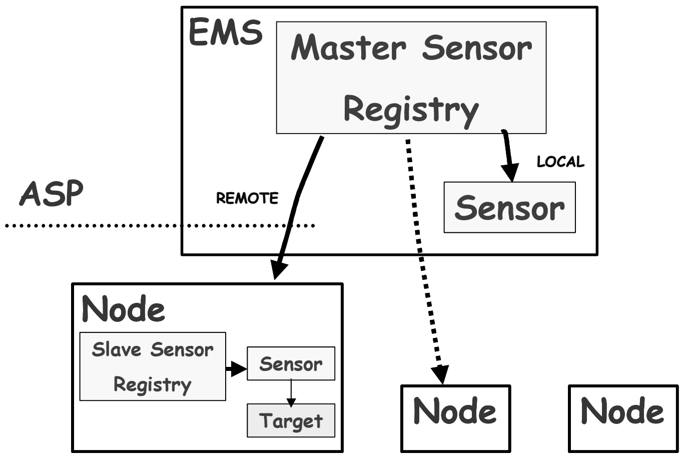


Figure 52. The Main EMS Monitoring System Components

Figure 52 shows how the remaining monitoring components are installed, taking into account such configuration information. Remote sensor deployment requires the previous installation in the node of a slave sensor registry, which from that moment takes control of the sensor instantiation, under the master sensor registry supervision. Once in the active node, the slave sensor registry is connected to the event channel and subscribed to receiving filters whose destination fits the particular node in which it is hosted.

From the point of view of the node management framework component model, the slave sensor registry assumes the Sensor Manager role, being responsible for the creation and management of the sensors. The monitoring process is completed when the sensor, after acquiring and processing the requested information, sends it to the event channel, where it will be distributed exclusively to the subscribed PDPs. The following paragraphs provide further description of the main components involved in these interactions.

2.4.3.5.1 Monitoring PDP:

Figure 53 presents the class diagram representing the main relationships between the control layer components. The monitoring PDP makes decisions regarding the location and configuration of the sensors required for performing a given network or service measurement. For this purpose, the PDP uses the DataBaseAccessController to gather the monitoring policies from the LDAP policy repository. After evaluating them, it generates a Decision, which is responsible for translating the policy actions into an appropriate SensorConfiguration (PIB). Eventually the Decision delivers such PIB to the PEP.

Each EMS station hosts a single monitoring PDP, which may control several PEPs (represented by master and slave sensor registries) in an asynchronous way. Therefore the PDP is prepared to maintain session information on each request being served and to handle several simultaneous PDP-PEP sessions.

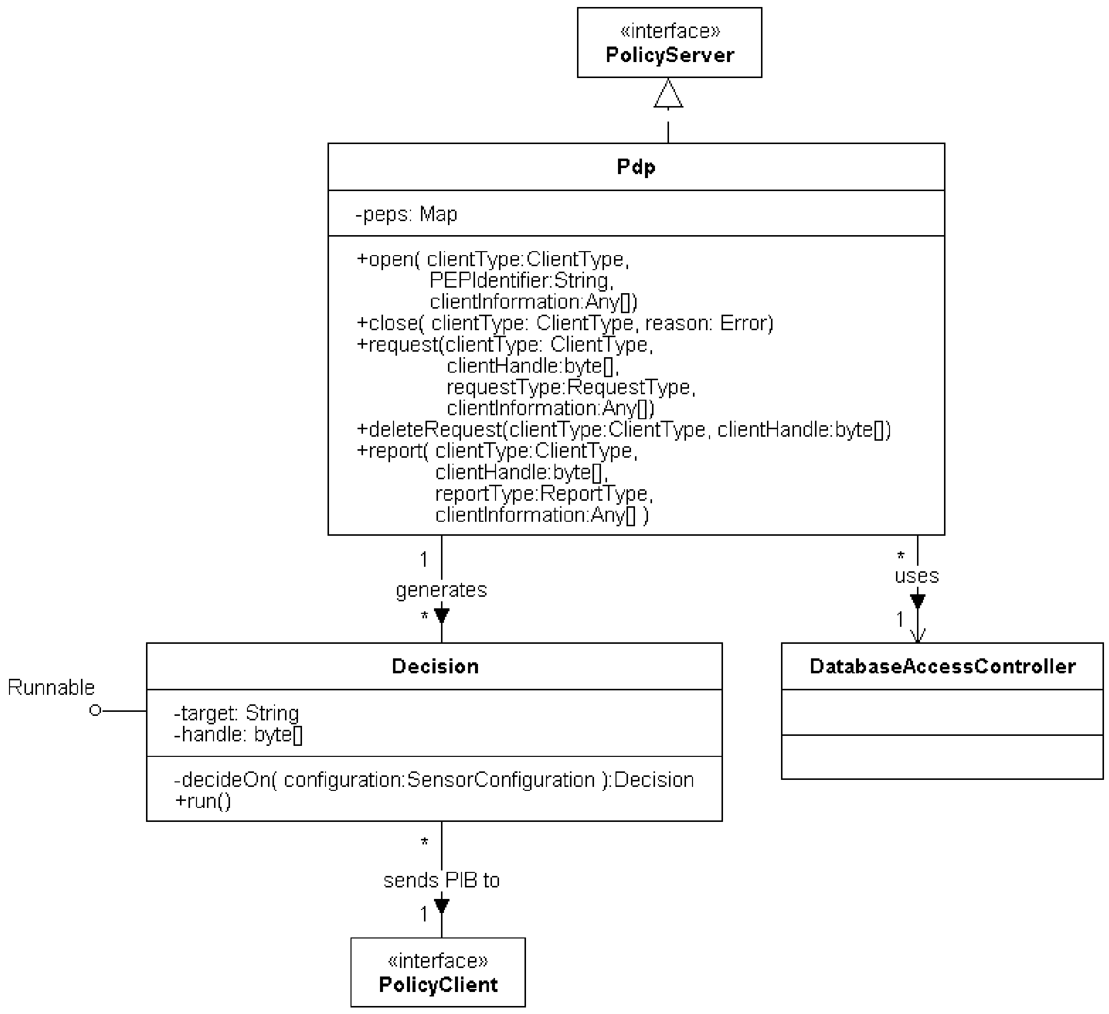

«interface»
PolicyServer
Pdp
-peps: Map
+open(clientType:ClientType, PEPIdentifier:String, clientInformation:Any[])
+close(clientType: ClientType, reason: Error)
+request(clientType: ClientType, clientHandle:byte[], requestType:RequestType, clientInformation:Any[])
+deleteRequest(clientType:ClientType, clientHandle:byte[])
+report(clientType:ClientType, clientHandle:byte[], reportType:ReportType, clientInformation:Any[])
1
generates
*
*
uses
1
Runnable
Decision
-target: String
-handle: byte[]
-decideOn(configuration:SensorConfiguration):Decision
+run()
DatabaseAccessController
*
sends PIB to
1
«interface»
PolicyClient

Figure 53. Control Layer Components

2.4.3.5.2 Monitoring PIB

The monitoring policy information is defined as an IDL structure used as a means to exchange specific monitoring configuration information through the COPS interface. Below we introduce its definition. The PIB contains fields identifying the target for the metering operation, and defining the strategy to capture the data and chain of manipulators that will be required to process them.

```
module pib {
         typedef struct ManipulatorDef {
           string name;
           short order;
           DynamicAny::NameValuePairSeq configuration;
         } tManipulatorDef;

         typedef struct CaptureDef {
           string name;
           string connector;
          DynamicAny::NameValuePairSeq configuration;
         } tCaptureDef;

         typedef sequence<CaptureDef> CaptureDefSeq;

         typedef struct SensorConfiguration {
           string targetID;
           boolean remote;
           CaptureDefSeq cdefSeq;
```

```
              ManipulatorDefSeq mdefSeq;
            } tSensorConfiguration;

          };
   }
```

2.4.3.5.3 Master Sensor Registry

As in the case of the monitoring PDP, there is only one instance of the master sensor registry per EMS station. As depicted in Figure 54, the SensorRegistry is actually the PolicyClient (PEP) in the policy based monitoring system. It receives filters from the SubscriptionBroker, representing the link with the extended notification channel, uses the FilterProcessor to process such filters and, based on the information gathered, it solicits a decision on how to perform the requested measurement operation.

Any operation involving the deployment of active code to the FAIN active nodes is carried out by the NetworkASPManager. This normally leads to the creation of a SlaveSensorRegistry in the active node which takes the role of a sensor manager, controlling the sensor lifecycle.

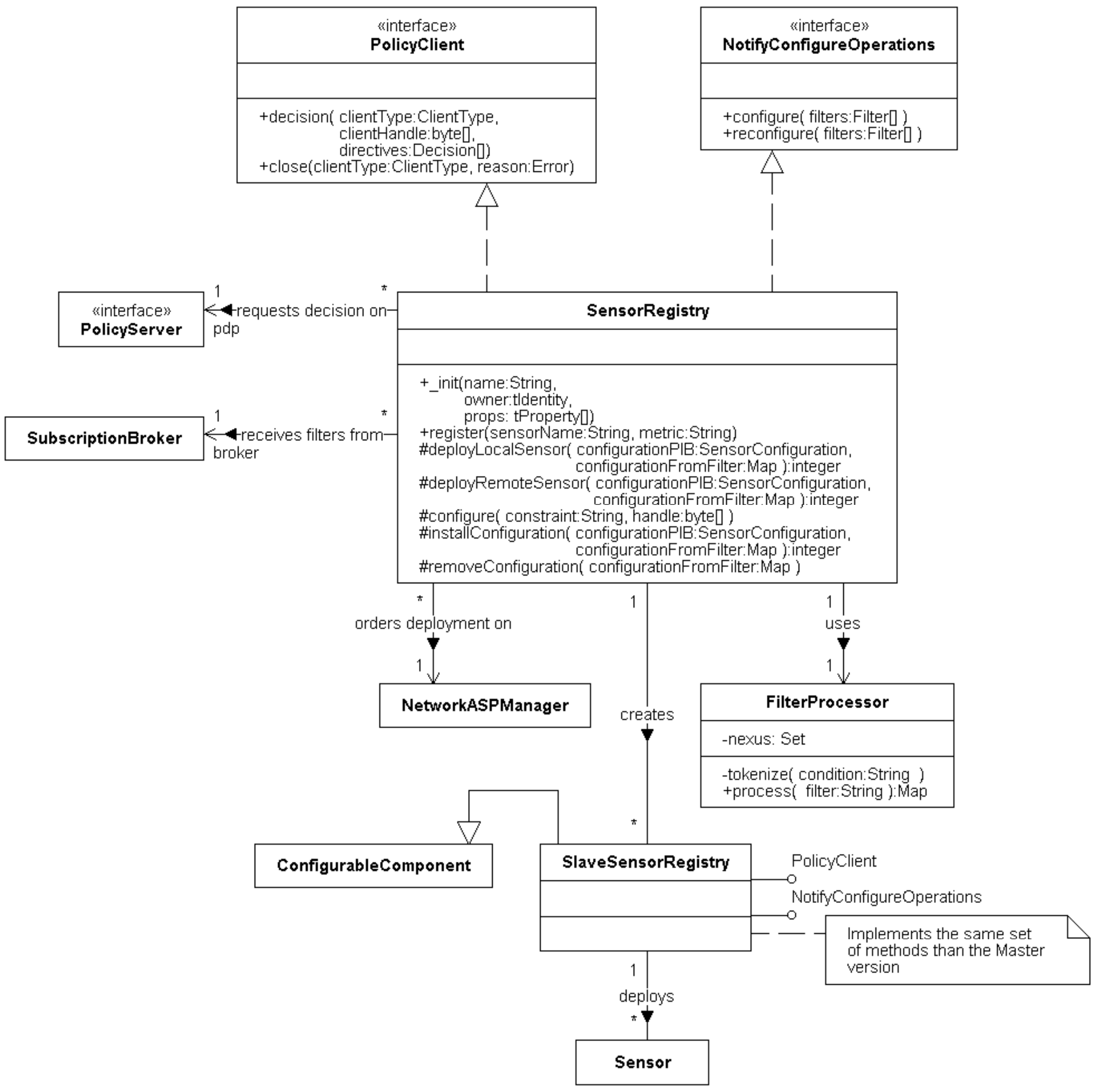

«interface»
PolicyClient
+decision(clientType:ClientType, clientHandle:byte[], directives:Decision[])
+close(clientType:ClientType, reason:Error)
«interface»
NotifyConfigureOperations
+configure(filters:Filter[])
+reconfigure(filters:Filter[])
«interface»
PolicyServer
requests decision on
pdp
SubscriptionBroker
receives filters from
broker
SensorRegistry
+_init(name:String, owner:tIdentity, props: tProperty[])
+register(sensorName:String, metric:String)
#deployLocalSensor(configurationPIB:SensorConfiguration, configurationFromFilter:Map):integer
#deployRemoteSensor(configurationPIB:SensorConfiguration, configurationFromFilter:Map):integer
#configure(constraint:String, handle:byte[])
#installConfiguration(configurationPIB:SensorConfiguration, configurationFromFilter:Map):integer
#removeConfiguration(configurationFromFilter:Map)
orders deployment on
NetworkASPManager
creates
uses
FilterProcessor
-nexus: Set
-tokenize(condition:String)
+process(filter:String):Map
ConfigurableComponent
SlaveSensorRegistry
PolicyClient
NotifyConfigureOperations
Implements the same set of methods than the Master version
deploys
Sensor

Figure 54. Acquisition Layer Components

2.4.3.5.4 Slave Sensor Registry

The SlaveSensorRegistry extends the ConfigurableComponent, and is fully integrated with the node management framework component model. The slave sensor registry assumes the role of a sensor manager on behalf of the master sensor registry, which controls each of the slaves.

It maintains connections with the policy repository, the monitoring PDP and the extended notification channel, being fully functional and autonomous during its operative state. However it should be noted that its influence is limited to the active node in which it runs. In fact, whilst the master is subscribed to receive every filter appearing in the channel, the slaves will receive only those going to the VEs actually instantiated in the hosting active nodes.

2.4.3.5.5 Sensor

A Sensor coordinates the data acquisition and manipulation activities at the lowest level. Each sensor interprets the SensorConfiguration information in order to create and join each of the monitoring building blocks. It creates each probe and assigns appropriate data capture strategies to them. Also it instantiates and chains the data manipulators in the specified order and connects the probes and manipulator as requested in the PIB.

2.4.3.5.6 Probe

A probe is a generic element that is responsible for gathering data using the capture strategies defined in the PIB. Additionally it manages event distribution among the data manipulator chains. The capture behaviour is encapsulated inside data capture strategies, where each strategy knows its set of configuration parameters. This model provides a higher degree of flexibility.

Indeed, the probe applies generic configuration mechanisms that allow previously unknown strategies to be followed. In order to achieve this, each strategy is questioned about its own configuration parameters and the list of available configuration values is traversed in order to match both of them based on the parameter name. When, as a consequence of adjusting one of the parameters, the probe detects the need for new configuration parameters, it requests the complete list again and the process is resumed.

2.4.3.5.7 The Data Manipulator Chain

Data manipulation is performed through the use of manipulation chains. Each manipulator is considered as a building block that performs a discrete operation on the data. After processing the data, the manipulator propagates the data to the next manipulator. Depending on the order of the manipulators different manipulations are possible. Additionally, several manipulation chains may be attached to the same probe, thus performing different processing operations on the same data.

3. ACTIVE SERVICE PROVISIONING

Active Service Provisioning, or ASP for short, is understood in the context of the FAIN project as a system for deploying active services in the FAIN network. In general, active service deployment is a process for making a service available in the active network so that the service user can use it. The deployment process is usually seen as a number of preparatory activities before the phase of the service operation. Typical activities include releasing the service code, distributing the service code to the target location, installing it and activating it.

Since the mid nineteen-nineties many efforts have been made to develop Active Networks technology to enable more flexible service provisioning in networks. By defining an open environment on network nodes this technology allows rapid deployment of new services which otherwise have a long lead-time and possibly require installation of new hardware.

The FAIN project follows an approach in which a number of existing and emerging active network technologies are integrated. With regard to deployment, FAIN proposes a novel approach to deploying services in heterogeneous active networks. In particular, the FAIN approach to deployment is characterized by the following:

- **On-demand service deployment support.** The ASP supports deployment of a service whenever it is needed. Service deployment may be explicitly requested by a service provider, by another service already deployed, or by a management component.
- **Component-based approach.** Deploying and managing high-level services requires an appropriate service model. While fully-fledged component-based service models are an integral part of many enterprise computing architectures (e.g. Enterprise JAVA Beans, CORBA Component Model, Microsoft's..NET), it is not the case in many approaches developed by the active networking community. The FAIN deployment framework is designed on top of a *component-based* service model similar to the CORBA Component Model. The service model is hierarchical in that service components may recursively include sub-components. This allows for a fine-grained service description and composition.
- **Network and node level architecture.** To deal with complexity of deployment in active networks, the Active Service Provisioning has been designed having separation of concerns in mind . The network-level ASP copes with network issues that include finding the nodes of the target environment for a given service considering topological service requirements as well as network link Quality of Service (QoS) requirements, for instance bandwidth. The node-level ASP, on the other hand, is concerned with node specific requirements, including technology and other service dependencies.
- **Integrated Service Deployment and Management.** The FAIN approach to service deployment is tightly integrated with FAIN service and network management. On one hand, the ASP depends on the service management framework which implements EE-specific deployment mechanisms, including installation and instantiation. On the other hand, the target environment in which the service is to be deployed is co-determined by the Network Management System. The target environment is defined to be a Virtual Active Network which is established by the FAIN Network Management System. The VAN is created by the management system according to the service requirements.
- **Selective code deployment.** Service code is distributed by selective downloading selected code modules from a code repository. The decision as to which code modules are needed is made by the ASP components at the target active nodes.
- **Support for heterogeneous services and networks.** The ASP has been designed to enable service deployment in heterogeneous networks. This is achieved by specifying a unified interface to the node capabilities and a unified notation for describing service specification and the implementation requirements. Whereas CORBA technology is used to define the unified API to the node, XML technology is used to define the unified service description.

The following sections give the details of the approach taken. Section 3.1 describes the final architecture of the Active Service Provisioning SystemThe interfaces of the ASP system are presented in section 3.3 and the design and implementation details of the ASP components are given in section 3.4.

3.1. ASP Final Architecture

The architecture of the ASP comprises two levels, the network level and the element level, as depicted in Figure 47.

On the network level the ASP consists of the Network ASP Manager working as the central access point of the ASP to other Non-ASP sub-systems. The Network Service Registry and the Network Service Repository are dedicated to service information storage and delivery:

- The *Network ASP Manager* has significantly extended functionality compared to the Network ASP Manager presented in D5 [1]. It is responsible for mapping services onto target nodes, and so provides mechanisms for evaluating nodes, focusing on service requirements. Furthermore it provides service inquiry and service management operations to retrieve static service information from the service descriptors or manage services respectively. It is solely able to parse network level service descriptors similar to the Service Creation Engine on element level for element level service descriptors. This service information partitioning over network level and element level is described in more detail in section 3.5.1.
- The *Network Service Registry* and the Network Service Repository are for service release and service update purpose. The Network Service Registry holds service descriptions in terms of service descriptors written in XML. These service descriptions are implementation-independent descriptions of the contents and internal structures of the service as well as the requirements and dependencies on other service or underlying infrastructures. On request giving the unique service name the Network Service Registry provides the descriptor of this service.
- The *Network Service Repository* stores the service code modules belonging to respective service descriptors and provides them on request for a given service code module name.

Combined, service descriptors and service code modules make up all service information to describe a service.

On the element level the ASP consists of the Node ASP Manager, which is the central access point for the ASP on the element level. The Node ASP Manager on the candidate nodes selected for deployment of service components is contacted by the network ASP manager . In addition the Code Manager, the Service Creation Engine, the Local Service Registry, the Local Service Repository and the Reconfiguration Manager make up the element level ASP:

- The *Node ASP Manager* delegates requests from network level to dedicated components of the element level ASP. Access to the element level ASP is only permitted through the Node ASP Manager.
- The *Code Manager* has unique knowledge about installed services and uses this knowledge during the service installation and service instantiation process. It is the coordinating component on the element level for the service code module fetching mechanism. It becomes active when requested by the Service Creation Engine.
- The *Service Creation Engine* is the key component of the element level ASP which processes the EE-independent part of the element level service descriptor. To perform the EE-dependent part of the service deployment process it cooperates with the deployment infrastructure of available Execution Environments.

- The *Local Service Registry* and the *Local Service Repository* are the counterparts of the Network Service Registry and the Network Service Repository. Basically they are caches for their network level counterparts as they fetch service descriptors and service code modules on request if not already fetched and available locally.
- A new component on the element level compared to the design presented in D5 is the *Reconfiguration Manager*. It focuses on service specific resource monitoring on the element level, in cooperation with the Node Manager of the Node Framework. It registers callback so that it can signal the need for a dynamic service reconfiguration to the Network ASP Manager, which then communicates with the Network Management System to get a decision whether a service reconfiguration should take place or not.

Figure 55, shows all the components of the ASP architecture. Whereas the node ASP components are located on active nodes, the network ASP (on the left of the figure) appears only once in the whole network. The node ASP communicates with Node Management Framework when requesting the node capabilities and installing service components. The network ASP is contacted by the Network Management System on behalf of the Service Provider that wants to deploy its services. The solid lines with arrows show the control flow when processing the deployment request of a service provider. The dashed lines show how the service code and service meta-data is transported to the active node during the deployment process. The details of this process are given in later sections.

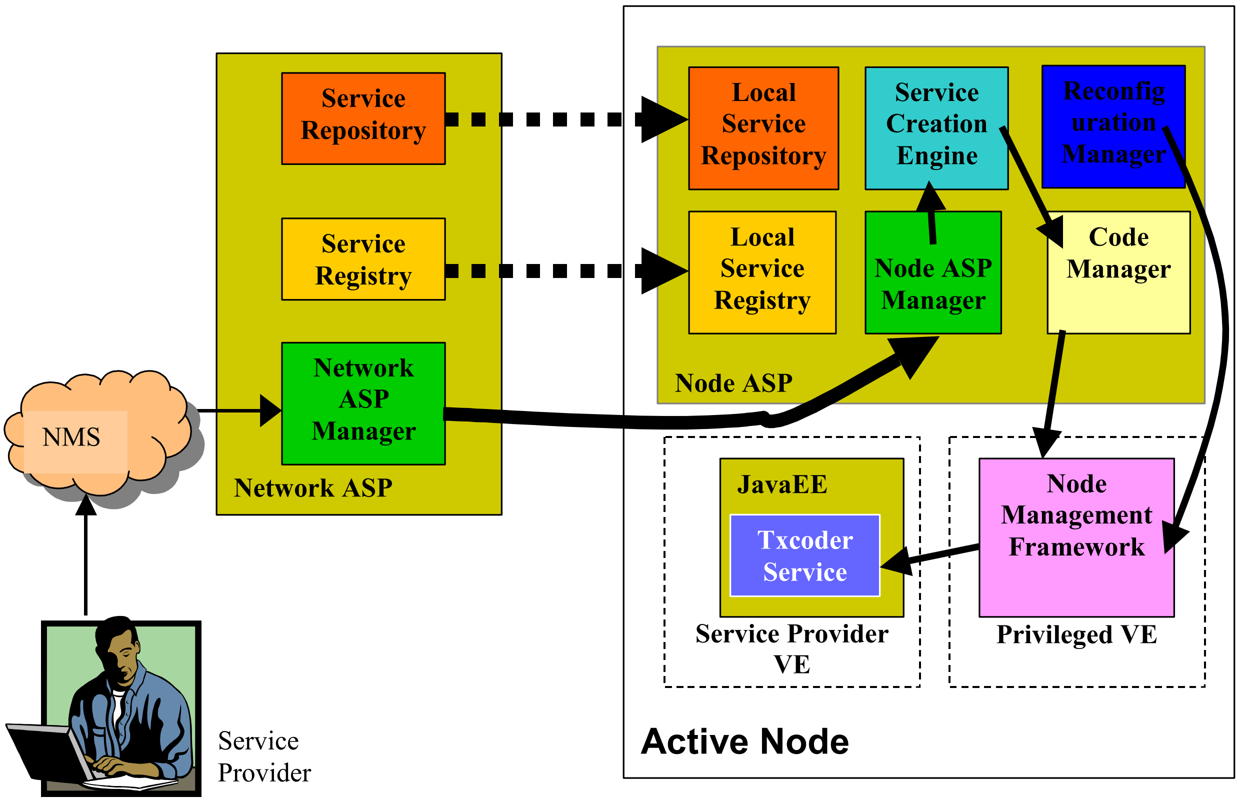


Figure 55. ASP Overview

3.2. ASP Functionalities

Figure 56 shows the functional range of the ASP, presenting the main use cases of the ASP. To obtain a deep understanding of the goals, tasks and requirements, the definition of the actors both from the FAIN enterprise model [3] and added later during the design phase are summarised in section 3.2.1. Complete ASP use cases [5] are described below in section 3.2.2.

3.2.1 Actors

In this context there are two types of actor, primary and supporting, each showing its own distinct behaviour. Primary actors use the services of the system under discussion (SuD) to fulfil user goals. They are defined in order to find the user goals that drive the use cases. Supporting actors provide services to the SuD. They are defined in order to identify external interfaces and protocols. The following actors have been identified for the ASP:

The primary actor is:

Service Provider, or SP for short, composes services that include active components and deploys these components in the network via the Active Service Provisioning, and offers the resulting service to Consumers. The service provider is responsible for releasing and withdrawing a service which includes a service version update or a complete remove of the service from specific nodes or from the complete active network respectively. Furthermore, the SP may be represented by the FAIN Network Management System with regard to initiation of service deployment or service reconfiguration.

The supporting actors are:

Active Network Service Provider, or ANSP for short, provides facilities for the deployment and operation of the active components into the network. Such facilities come in the form of an active middleware, support of new technologies. ANSP is represented by Active Nodes which are the target environment in context of deployment, which means that services may be deployed in these nodes and use the node resources made available to them by the ANSP.

Network Infrastructure Provider, or NIP for short.

Service Component Provider, or SCP for short.

3.2.2 Use Case Diagrams

This section describes the functionalities of the ASP system in terms of use cases. The following use cases have been identified:

- *Releasing a service.* The Service Provider who decides to offer his service in the active network has to release it in the active network. The service is released by making the service meta-information and service code modules available to the ASP system.
- *Deploying a service.* After the service is released in the network, the Service Provider may want to deploy his service so that it can be used by a given service user. This means finding target nodes that are most suitable for the given service installation, determining a mapping of the service components to the available Execution Environments of the target node, downloading the appropriate code modules, and finally installing and activating them.
- *Removing a service.* The Service Provider may request to remove a deployed service from the environment in which it was deployed. The ASP identifies the installed service components and removes them from the Execution Environments (EE's) of the target environment.
- *Withdrawing a service.* A service released in the active network may be withdrawn so that is no longer available to be deployed. The ASP removes the service meta-information and discards the service code modules.

- *Reconfiguring a service.* Changes of the current configuration of a service may be requested. Requesters of reconfigurations could be the Service Provider (SP), the Network Management System (NMS), the Element Management System (EMS) or the Active Network Service Provider (ANSP). Reconfiguration may include modifying component bindings, deploying additional service components or redeploying components that have been already deployed.
- *Updating a service.* The Service Provider may announce a new version of an already released service to an active network. The service code and metadata of the new version of the service have to replace the code and metadata of the old (updated) version.

Figure 56 presents the identified use cases, their interdependencies as well as the actors interacting with them. The following sub-sections provide the detailed descriptions of these use cases. The description of the use cases is structured according to a use case description template presented in [5].

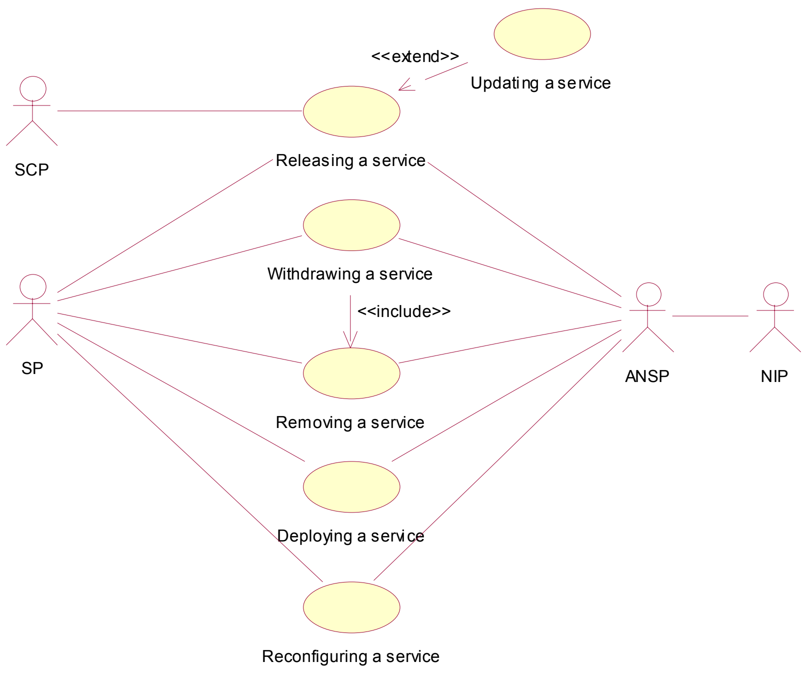


Figure 56. Main Use Case of the ASP

3.2.2.1 Releasing a service / Updating a service

Figure 57 depicts use case diagram “Releasing a service/Updating a service”.

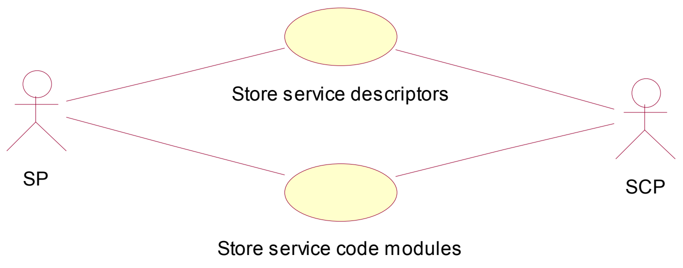


Figure 57. Releasing a service / Updating a service - Use case diagram

Primary actor: Service Provider

Stakeholders and interests:

- Service Provider: Wants to offer a service in an active network.
- Service Component Provider: Wants its services to be applied in an active network.

Preconditions: Service descriptors for network and element level as well as dedicated service code modules are available.

Postconditions: Service descriptors for network level and element level are stored in the Network Service Registry. Respective service code modules are stored at the Network Service Repository.

Basic flow:

1. Service Provider obtains all service components from the Service Component Provider: The service components are: service descriptors for network level and element level and the respective service code modules.
2. Service Provider requests storage of the service descriptors from the Network Service Registry and storage of the service code modules from the Network Service Repository.
3. Network Service Registry stores the service descriptors. Network Service Repository stores the service code modules.
4. Service Provider gets feedback about a successful storage of all service components from the Network Service Registry and the Network Service Repository.

Extensions:

At any time, System fails:	Since the service descriptors and service code modules are stored persistently, usually a restart of the Network Service Registry and Network Service Repository is enough to make them work again. If service components are not stored yet the storage of the service parts have to be requested again.

3a. In case of a failure in storing the service descriptors or the service code modules the basic flow has to be repeated for the respective service.

Special Requirements: None

Data Variations List:

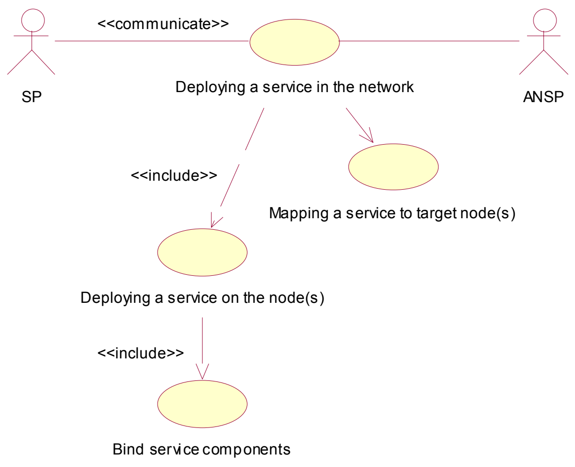
1a. Network level service descriptors have to be consistent to the network level service descriptor schema. Element Level service descriptors have to be consistent to the element level service descriptor schema.

Frequency of Occurrence: Releasing a service: At least once per service, at latest before deployment.

Updating a service: If any nearly continuous after a service is released

3.2.2.2 Deploying a service

Figure 58 depicts use case diagram "Deploying a service".



<<communicate>>
SP
Deploying a service in the network
ANSP
<<include>>
Mapping a service to target node(s)
Deploying a service on the node(s)
<<include>>
Bind service components

Figure 58. Deploying a service - Use case diagram

Primary actor: Service Provider

Stakeholders and interests:

- Service Provider: Wants to deploy a service in an active network.
- Active Network Service Provider: Offers facilities for the deployment and operation of service components in an active network.

Preconditions: Releasing a service

Postconditions: Service is deployed in an active network.

Basic flow:

1. Service Provider requests the deployment of a service in an active network from the ASP.
2. The network level ASP performs a service requirements to target environments mapping to find matching target nodes to component requirements of a service. This mapping of service requirements to target environments is done in cooperation with the Service Creation Engine (SCE) on element level.
3. Deployment of service components to the found target nodes is initiated.
4. Download of element level service descriptors at the found target nodes.

5. The element level service descriptor is processed by the SCE to resolve dependencies to map service component names to implementations suitable to local node environments.
6. The Code Manager is ordered by the SCE to download service code modules of determined implementations.
7. The Code Manager installs, configures and instantiates downloaded service code modules right after their download.

Extensions:

2a. If mapping of service requirements to target environments fails, the deployment process is cancelled. The service deployment requester is notified about this failure.

5a. If no implementations are found for service component names during the dependency resolution by the SCE, the service components can not be deployed onto these nodes. The deployment process has to be cancelled. The service deployment requester is notified about this failure.

Special Requirements: None

Data Variations List: None

Frequency of Occurrence: Could be nearly continuous considering different SP's requesting deployments of different services.

3.2.2.3 Reconfiguring a service

3.2.2.3.1 Dynamic Reconfiguration of a service

Figure 59 depicts use case diagram "Dynamic Reconfiguration".

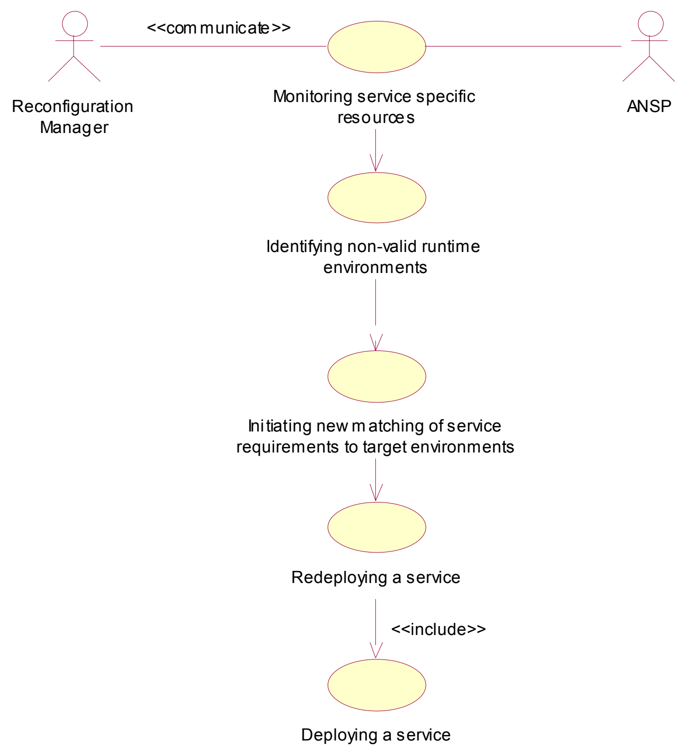

Reconfiguration Manager
<<communicate>>
Monitoring service specific resources
ANSP
Identifying non-valid runtime environments
Initiating new matching of service requirements to target environments
Redeploying a service
<<include>>
Deploying a service

Figure 59. Dynamic Reconfiguration – Use case diagram

Primary actor: Reconfiguration Manager

Stakeholders and interests:

- Reconfiguration Manager: Wants to alert an inconsistency between service requirements and respective service runtime environments to the Network ASP Manager.Active Network Service Provider: Offers facilities for the deployment and operation of service components in an active network.

Preconditions: An inconsistency exists between service requirements and respective service runtime environments

Postconditions: Runtime environments match service requirements

Basic flow:

1. Monitoring of service specific resources by the Node Manager of the Node Framework on each node.
2. Node Manager alerts the Resource Manager on its node in case of thresholds of service specific resources are reached.
3. Resource Manager alerts the Network ASP Manager about this mismatch of service requirement to runtime environment.
4. Network ASP Manager forward this alert to the NMS as it is the NMS that decides on a re-matching of requirements of a service to target environments.

5. The NMS initiates a re-matching of requirements of a service to target environments.
6. The Network ASP performs this re-matching process and initiates the deployment process of the service or its components respectively.

Extensions:

5a. The NMS decides not to deploy this service again.

6a. The Network ASP finds out that no target environment fits the service requirements. It delivers this result of its re-matching process back to the NMS which decides about further steps.

Special Requirements: None

Data Variations List: None

Frequency of Occurrence: Could be nearly continuous.

3.2.2.3.2 Management controlled Reconfiguration of a service

Figure 60 depicts use case diagram "Management controlled Reconfiguration".

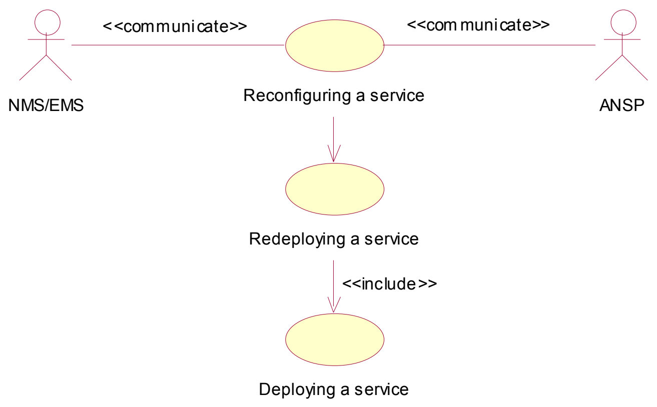


Figure 60. Management controlled Reconfiguration

Primary actors: NMS, EMS, Active Network Service Provider

Stakeholders and interests:

- NMS: Wants to reconfigure management services like Policy Enforcement Point (PEP) services and Policy Decision Point (PDP) services.
- EMS: Wants to reconfigure management services like PEP services and PDP services.
- Active Network Service Provider: Wants to deploy its PDP/PEP service in the network. Offers facilities for the deployment and operation of service components in an active network.

Preconditions: Already deployed PDP/PEP services have to be reconfigured to adapt to changed environments.

Postconditions: PDP/PEP services fit the changed environments.

Basic flow:

1. NMS/EMS requests a reconfiguration of PDP/PEP services in a way that these PDP/PEP services should run on different nodes than they are running on now. NMS/EMS specify on which nodes which PDP/PEP service should run.
2. Network ASP Manager deploys the PDP/PEP services to the specified nodes without checking whether a service deployment would be successful or not.

Alternative flow:

1. ANSP wants to deploy its PDP/PEP services in the network.
2. Network ASP Manager deploys the PDP/PEP services to the specified nodes without checking whether a service deployment would be successful or not.

Extensions, valid for both flows:

2a. Due to the fact that not enough resources are available for a PDP/PEP service the Network ASP Manager receives an error message from the Node ASP Manager and passes this negative response to its request to the NMS/EMS.

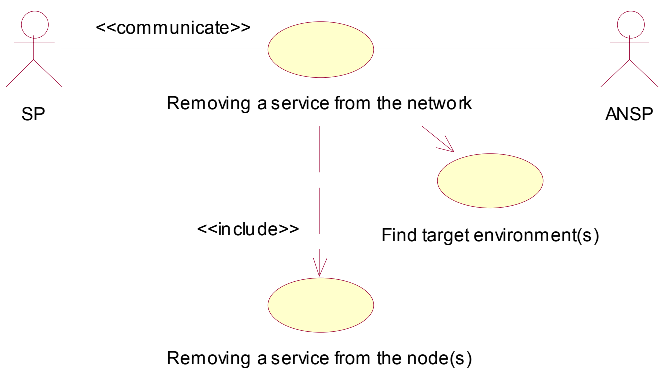
Special Requirements: None

Data Variations List: None

Frequency of Occurrence: Could be nearly continuous.

3.2.2.4 Removing a service

Figure 61 depicts use case diagram "Removing a service".



<<communicate>>
SP
Removing a service from the network
ANSP
<<include>>
Find target environment(s)
Removing a service from the node(s)

Figure 61. Removing a service - Use case diagram

Primary actor: Service Provider

Stakeholders and interests:

- Service Provider: Wants to remove a service and its components respectively from environments it was deployed in.
- Active Network Service Provider: Offers facilities for the deployment and operation of service components in an active network.

Preconditions: Deploying a service

Postconditions: Service and its component respectively are removed from environments it was deployed in.

Basic flow:

1. Service Provider requests the removal of a service from an active network.
2. Search for environments the service was deployed in.
3. Resolve dependencies to map service component names to implementations suitable to local node environments. Instances of the respective service code modules corresponding to the found implementations are removed and the service code modules are uninstalled from the specific execution environments.

Extensions:

2a. If the search for environments the service was deployed in is not successful for some reason, then the requester receives an error message including the reason.

3a. If de-instantiations or de-instantiations fail for some reason the requester receives an error message including the reason.

Special Requirements: None

Data Variations List: None

Frequency of Occurrence: If at all, once per deployed service.

3.2.2.5 Withdrawing a service

Figure 62 depicts use case diagram "Withdrawing a service".

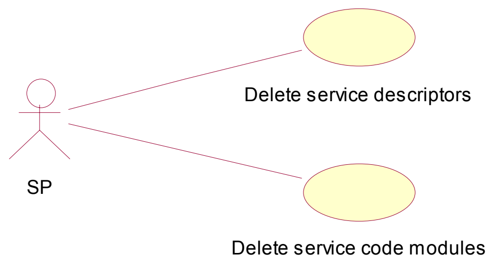


Figure 62. Withdrawing a service - Use case diagram

Primary actor: Service Provider

Stakeholders and interests:

- Service Provider: Wants to withdraw a service from an active network.
- Active Network Service Provider: Offers facilities for the deployment and operation of service components in an active network.

Preconditions: Releasing a service

Postconditions: Service is completely unknown to the active network.

Basic flow:

1. Service Provider requests deletion of the service descriptors from the Network Service Registry and storage of the service code modules from the Network Service Repository. Existing service descriptors at element level counterparts are also deleted so no service parts are available in the active network anymore.
2. Service descriptors are deleted from Network Service Registry and its element level counterpart. Service code modules are deleted from Network Service Repository and its element level counterpart.
3. Service Provider gets feedback about a successful deletion of all service parts from the Network Service Registry and the Network Service Repository.

Extensions:

At any time, System fails: Since the service descriptors and service code modules are stored persistently usually a restart of the Network Service Registry and Network Service Repository is enough to make them work again. If service parts are still stored the deletion of the respective service parts have to be requested again.

3a. In case of a failure in deleting the service descriptors or the service code modules the basic flow has to be repeated for the respective service.

Special Requirements: None

Data Variations List: None

Frequency of Occurrence: If at all, once per released service.

3.3. Application Programming Interface

The Network ASP as the central access point of the ASP offers an external Application Programming Interface (API) for service inquiry and service management. The IDL specification of this external API is described in detail below.

3.3.1 Service Inquiry Interface

Operations on the service inquiry interface are dedicated for service information delivery. This service information is only extracted from network level service descriptors since the Network ASP has exclusive knowledge of network level service information. Element level service information is not delivered in any way outside the ASP because this service information is for exclusive use within the ASP.

A network level service descriptor should contain at least one service component. Operation `listServiceComponentNames()` delivers the names of the service components mentioned in the network level service descriptor:

```
ServiceComponentNames listServiceComponentNames
(in ServiceName serviceName)
  raises (ServiceNotFound);
```

Every service component has specific runtime requirements. They can be extracted from the network level service descriptor and delivered to the requester using operation `getServiceComponentRequirements()`:

```
Properties getServiceComponentRequirements
(in ServiceName serviceName, in ServiceComponentName componentName)
  raises (ServiceNotFound, ServiceComponentNotFound);
```

Operation `getServiceComponentLinkRequirements()` extracts and delivers link requirements between two given services:

```
LinkRequirements getServiceComponentLinkRequirements
(in ServiceName serviceName, in ServiceComponentName from,
 in ServiceComponentName to)
  raises (ServiceNotFound, ServiceComponentNotFound);
```

The topology of a given service can be requested using operation `getServiceTopologyRequirements()`:

```
ServiceTopologyInfo getServiceTopologyRequirements
(in ServiceName serviceName)
  raises (ServiceNotFound, ServiceComponentNotFound);
```

3.3.2 Service Management Interface

Compared to static service information delivered by operations on the service inquiry interface, the service management interface offers operations using this service information to fulfil different tasks, i.e. service release or service deployment.

For service release and service withdrawing the Network ASP offers operations `registerService()` and `unregisterService()`. Registration includes the announcement of service descriptors to the Network Service Registry and the announcement of service code modules to the Network Service Repository:

```
void registerService
(in ServiceName serviceName,
 in ServiceDescriptor serviceDescriptor,
 in URLs serviceURLs,
 in org::ist_fain::tIdentity who)
  raises (ServiceRegistrationFailed);

void unregisterService
(in ServiceName serviceName,
 in org::ist_fain::tIdentity who)
  raises (ServiceNotFound);
```

For given VANPaths the best fitting candidate for the deployment of a service is determined by operation `calculateBestCandidate()`:

```
VANPath calculateBestCandidate
(in ServiceName serviceName,
 in VANPaths vanPaths,
 in org::ist_fain::tIdentity who)
  raises (ServiceNotFound, ServiceMappingFailed);
```

After one VANPath was chosen to deploy a service, operation `createServiceInstance()` realizes the deployment of this service:

```
ServiceReferenceInfos createServiceInstance
(in ServiceName serviceName,
 in org::ist_fain::node::management::tPropertyList aConfiguration,
 in VANPath vanPath,
 in org::ist_fain::tIdentity who)
  raises (ServiceNotFound, InstantiationFailed);
```

If a running service is not needed any more, it can be removed from the environment in which it was deployed. The installed service components are identified and removed from the Execution Environments of the target environment:

```
       void removeServiceInstance
       (in ServiceInstanceID serviceInstance,
        in org::ist_fain::tIdentity who)
         raises (ServiceInstanceNotFound, RemovalFailed);
```

If a service could not be released before requesting its deployment, operation `deployServiceComponent()` can be used. This operation will implicitly perform the service release right before initiating the service deployment for a given VANPath:

```
       ServiceReferenceInfos deployServiceComponent
       (in ServiceName serviceName,
        in org::ist_fain::node::management::tPropertyList aConfiguration,
        in ServiceDescriptor descriptor,
        in URL codeModuleURL,
        in VANPath vanPath,
        in org::ist_fain::tIdentity who)
              raises (InstallationFailed);
```

For the distribution of a service code module to a node of a VAN, dependent on where an instance of a specific service is running, operation `addServiceCodeModule()` is needed:

```
       CodeModuleRef addServiceCodeModule
       (in ServiceInstanceID serviceInstance,
        in CodeModuleID codeModuleID,
        in VANNodeID vanNodeID,
        in org::ist_fain::tIdentity who)
         raises (ServiceInstanceNotFound, CodeModuleNotFound, InstallationFailed);
```

The counterpart of operation `addServiceCodeModule()` to remove the distributed service code module from the specified VAN node where an instance of a given service is running is operation `removeServiceCodeModule()`:

```
       void removeServiceCodeModule
       (in ServiceInstanceID serviceInstance,
        in CodeModuleID codeModuleID,
        in VANNodeID vanNodeID,
        in org::ist_fain::tIdentity who)
         raises (ServiceInstanceNotFound, CodeModuleNotFound, RemovalFailed);
```

To receive a list of ServiceComponentID’s of running service components of a specific service, operation `listRunningServiceComponents()` is needed:

```
       ServiceComponentIDs listRunningServiceComponents
       (in ServiceInstanceID serviceInstance)
         raises (ServiceInstanceNotFound);
```

To receive only the ServiceComponentID of one running service component within a specific service, operation `getServiceComponentID()` is needed:

```
       ServiceComponentID getServiceComponentID
       (in ServiceInstanceID serviceInstance,
        in ServiceComponentName componentName)
         raises (ServiceInstanceNotFound, ServiceComponentNotFound);
```

If only the IP address of a node and the name of a service running on this node are known, operation getServiceRef() delivers the reference of this service:

```
ServiceReferenceInfo getServiceRef
(in ServiceName serviceName,
 in NodeIPAddress ipAddress)
  raises (ServiceNotFound, IPAddressNotFound);
```

To receive the ServiceComponentInfo structure for a specific service component, operation getServiceComponentInfo() can be used.

```
ServiceComponentInfo getServiceComponentInfo
(in ServiceComponentID componentID)
  raises (ServiceComponentInstanceNotFound);
```

A ServiceComponentInfo is structured as follows:

```
struct ServiceComponentInfo {
 org::ist_fain::node::management::iComponentInitial initial;
};
```

If a running service instance exists, a service component can be instantiated on the same node as the running service instance using operation addServiceComponent(). Among other parameters a VANPath in which the service instance can be found is needed to call this operation:

```
        ServiceReferenceInfos addServiceComponent
        (in ServiceInstanceID serviceInstance,
         in ServiceComponentName componentName,
         in org::ist_fain::node::management::tPropertyList aConfiguration,
         in VANPath vanPath,
         in org::ist_fain::tIdentity who)
          raises (ServiceInstanceNotFound, ServiceComponentNotFound,
InstallationFailed);
```

The IP address of a node and the Virtual Environment ID a component should be deployed in is needed mainly to use operation addComponentToNode() to add a service component instance on a specific node given by the IP address:

```
        ServiceReferenceInfos addComponentToNode
        (in ServiceComponentName componentName,
         in NodeIPAddress ipAddress,
         in org::ist_fain::node::management::tPropertyList aConfiguration,
         in VeID veID,
         in org::ist_fain::tIdentity who)
          raises (ServiceInstanceNotFound, ServiceComponentNotFound,
InstallationFailed);
```

To remove a service component instance from a given service instance, operation removeServiceComponent() is needed:

```
void removeServiceComponent
(in ServiceComponentID componentID,
 in org::ist_fain::tIdentity who)
  raises (ServiceComponentInstanceNotFound, RemovalFailed);
```

For management controlled reconfiguration it is necessary to bind and unbind service components. Operations bindServiceComponents() and unbindServiceComponents() are needed for such binding or unbinding processes respectively.

```
void bindServiceComponents
(in ServiceComponentID first, in ServiceComponentID second,
 in org::ist_fain::tIdentity who)
  raises (ServiceComponentInstanceNotFound, BindingFailed);

void unbindServiceComponents
(in ServiceComponentID first, in ServiceComponentID second,
 in org::ist_fain::tIdentity who)
  raises (ServiceComponentInstanceNotFound, UnbindingFailed);
```

For bind a given service component instance to a specified WP3 port, operation bindServiceComponentPort() is offered. To unbind a formerly bind service component from a specified WP3 port, operation unbindServiceComponentPort is used:

```
void bindServiceComponentPort
(in ServiceComponentID componentID,
 in org::ist_fain::node::management::tPortName cPortName,
 in org::ist_fain::node::management::tPort otherPort,
 in org::ist_fain::tIdentity who)
  raises (ServiceComponentInstanceNotFound, BindingFailed);

void unbindServiceComponentPort
(in ServiceComponentID componentID,
 in org::ist_fain::node::management::tPortName cPortName,
 in org::ist_fain::node::management::tPort otherPort,
 in org::ist_fain::tIdentity who)
  raises (ServiceComponentInstanceNotFound, UnbindingFailed);
```

3.4. Service Description

To deploy a service according to its needs, a notion of describing these requirements is needed. In the FAIN project, a service descriptor is used. It includes basic information about the service as well as the deployment requirements of the service.

Because of the complexity of the information needed for the service to be deployed, a structured notion is needed to handle this information. In the FAIN project, the XML language was chosen to define a service descriptor.

Furthermore, a service descriptor is divided into two parts: network level and node (element) level. There are two main reasons of including network and node levels in the service descriptor.

- *Separation of concerns***.** The ASP was designed using a two layer architecture to deal with different issues at different layers. Whereas the network level ASP is concerned with network issues, including service distribution on network nodes, the node level ASP deals with issues local to the node, like choosing the target execution environment and resolving service component dependencies. It is thus natural to separate these concerns and deal with them using separate data structures, i.e. in the context of the ASP - service descriptors.
- *Re-use the node ASP***.** The previous work in FAIN resulted in the implementation of the node ASP. To maximally re-use this software, a design decision was made to specify another network level service descriptor. This descriptor is based on the XML schema used for the node level service descriptor.

The next sections are structured as follows: Section 3.4.1 introduces the basic concepts used for describing a service for deployment purpose. Section 3.4.2 gives the details of the network level service descriptor, whereas section 3.4.3 describes the node level service descriptor.

3.4.1 Basic Concepts

The following concepts are helpful when understanding the way a FAIN service is structured for the deployment purposes.

Service Deployment – a process that involves fetching, installing and loading all the *service components*' *code modules* into their *Target Environments*.

Service – a unit of functionality that a service provider wants to offer to the customers. In terms of deployment, it is a service component (without any unresolved inter-code module independencies).

Service Component – a unit of deployment. It consists of:

- service descriptor kept in the service registry and possibly in its local counterpart.
- optional reference to a code module kept in the service repository.

There are three classes of service component that differ in the terms of whether they consist of service sub-components and whether they directly refer to a code module.

(Simple) Implementation – a service component without any dependencies. It contains just a reference to a code module.

Compound Implementation – a service component consisting of sub-components and having a reference to a code module.

Abstract Implementation - a service component consisting of sub-components and having no reference to a code module.

Target environment – the runtime environment that the service component is to be installed. In FAIN, the target environment is defined by a virtual execution (resource management) and an execution environment.

Code Module is a file with the code. The contents of the file are obscure to the ASP. It may contain however some EE-specific information, on how to deploy the code module, which is used by the EE configurator. A code module has the following phases of life cycle (also depicted in Figure 63) from the node perspective:

- It is **fetched** onto a local node from the service repository and may be kept in the local service repository for some time.
- It is **installed** into an EE instance or an EE type. This is done by configuring the running instances of the EE or changing the EE templates that are used to instantiate new EE in which the code modules are to be installed.
- It is **loaded** into an EE instance. The module is either dynamically loaded into the running EE instance or loaded by a new EE instance is started.

The code module may be also unloaded and changes to the installed state. It may be also uninstalled for a certain EE instance on demand. The Code Manager keeps information about installed code modules.

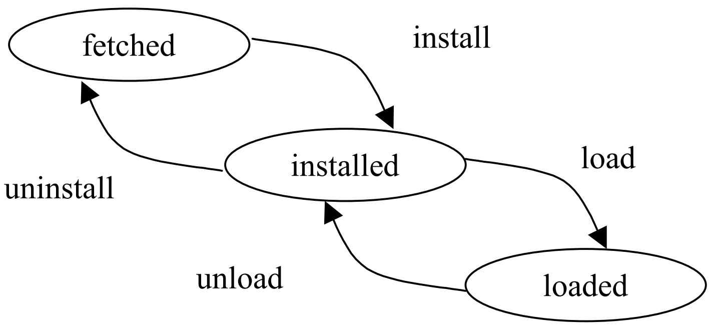


Figure 63. The life cycle phases of the code module

3.4.2 Network-level Service Descriptor

The Network ASP which is exclusively responsible for processing network level service information extracts this service information from network level service descriptors.

A network level service descriptor can specify the following:

Service Description:

- Name of the service (SERVICE_NAME)
- Identifier for the service (SERVICE_ID)
- Name of the provider of the service (PROVIDER)
- Version of the service (VERSION)
- Signature of the service (SIGNATURE)
- Classifier for the service (CLASS)
- License for the service (CLASS)

Service Component:

- Name of the service component (NAME)
- Name of the service instance to differentiate between service components of name NAME (INSTANCE_NAME)
- Location of the service component relative to nodes fulfilling specific roles (LOCATION/RELATIVE/NODE_ROLE)

The number of service components specified in a network level service descriptor is not limited.

Connections between service components running on different nodes are not yet specified in network level service descriptors since this implies the ability of the Node Framework to do inter-node binding operations, which is not yet supported.

```
  <NETWORK_SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="/home/mos/FAIN_network_level_descriptor.xsd"
xsi:type="NETWORK_SERVICE">
        <DESCRIPTION>
                <SERVICE_NAME>extended_transcoder</SERVICE_NAME>
                <SERVICE_ID>extended_transcoder_pure_java</SERVICE_ID>
                <PROVIDER>FT</PROVIDER>
                <VERSION>0.2</VERSION>
                <SIGNATURE>0x1234ab006f8b11fa8c</SIGNATURE>
                <CLASS>economy</CLASS>
                <LICENSE>0.2</LICENSE>
        </DESCRIPTION>
        <SERVICE_COMPONENT>
                <NAME>duplicator</NAME>
                <INSTANCE_NAME>d1</INSTANCE_NAME>
                <LOCATION>
                        <RELATIVE>
                               <NODE_ROLE>ingress
                               </NODE_ROLE>
                        </RELATIVE>
                 </LOCATION>
        </SERVICE_COMPONENT>
        <SERVICE_COMPONENT>
                <NAME>TXengine</NAME>
                <INSTANCE_NAME>tx1</INSTANCE_NAME>
                <LOCATION>
                        <RELATIVE>
                               <NODE_ROLE>egress
                               </NODE_ROLE>
                        </RELATIVE>
                 </LOCATION>
        </SERVICE_COMPONENT>
</NETWORK_SERVICE>
```

Figure 64. Example of a network level service descriptor

3.4.3 Node Level Service Descriptor

The service descriptor describes the node level meta-information of the service. The first part of the service descriptor holds information about the service component developer or provider and its functionality. The second part is dependent on the class of service component described. For a simple implementation this part contains a reference to a code module and identifies the target environment where the code module is to be installed. It also contains EE-specific information, which is used to perform EE-specific part of deployment process.

The service descriptor of an abstract implementation holds information about required sub-components and how they are to be bound to each other in order to perform the expected functionality. Finally, a compound implementation is a mixture of the two classes above, and hence contains both sets of information. The service descriptor is implemented in XML, which proved to be a very suitable technology for the task at hand. Furthermore, we developed an XML Schema to verify the structure and correct syntax of service descriptors.

Figure 64 gives a simple example of such a descriptor.. The service “transcoder” is composed of two sub-services” transcoder1” and” transcoder2”.

The first XML descriptor detailed the service descriptor for the global service (transcoder) and the second XML descriptor detailed the service descriptor for the sub-service “transcoder1”, which is a real implementation and a running service.

Transcoder.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by Matthias Bossardt
(ETH Zurich) -->
<SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Documents and Settings\bossardt\My
Documents\Projects\Chameleon\XML\chameleon.xsd" xsi:type="SPECIFICATION">
      <DESCRIPTION>
            <SERVICE_NAME>transcoder</SERVICE_NAME>
            <SERVICE_ID/>
            <PROVIDER>FT</PROVIDER>
            <VERSION>0.1</VERSION>
      </DESCRIPTION>
      <SUB_SERVICE>
            <SERVICE_NAME>transcoder1</SERVICE_NAME>
            <INSTANCE_NAME>t1</INSTANCE_NAME>
      </SUB_SERVICE>
      <SUB_SERVICE>
            <SERVICE_NAME>transcoder2</SERVICE_NAME>
            <INSTANCE_NAME>t2</INSTANCE_NAME>
      </SUB_SERVICE>
   </SERVICE>
```

Transcoder1.xml:

```
   <?xml version="1.0" encoding="UTF-8"?>
   <!-- edited with XML Spy v4.3 U (http://www.xmlspy.com) by Matthias Bossardt
(ETH Zurich) -->
   <SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Documents and Settings\bossardt\My
Documents\Projects\Chameleon\XML\chameleon.xsd" xsi:type="IMPLEMENTATION">
      <DESCRIPTION>
            <SERVICE_NAME>transcoder1</SERVICE_NAME>
            <SERVICE_ID/>
            <PROVIDER>FT</PROVIDER>
            <VERSION>0.1</VERSION>
      </DESCRIPTION>
      <PROPERTIES>
            <PROPERTY>
                  <KEY>mainClassName</KEY>

      <VALUE>org.ist_fain.services.transcoder1.TranscoderManager</VALUE>
            </PROPERTY>
            <PROPERTY>
                  <KEY>mainCodePath</KEY>
                  <VALUE>/usr/local/jmf-2.1.1/lib/jmf.jar:/usr/local/jmf-
2.1.1/lib/sound.jar:/usr/local/jmf-2.1.1/lib:code/demux.jar</VALUE>
            </PROPERTY>
            <PROPERTY>
                  <KEY>AdmissionTimeOut</KEY>
                  <VALUE>30000</VALUE>
            </PROPERTY>
      </PROPERTIES>
      <ENVIRONMENT>
            <EE_NAME>JVM</EE_NAME>
            <EE_VERSION>1.3.1</EE_VERSION>
      </ENVIRONMENT>
      <CODE xsi:type="CODE_LOCATION">
            <CODEBASE>jvm.transcoder1.FT.transcoder1.jar</CODEBASE>
      </CODE>
   </SERVICE>
```

Figure 65. Example of a node level service descriptor

3.5. ASP Components

In this section, the design and implementation details of the ASP components are given. Whereas the functionality that has not changed since publishing [1] is briefly summarised, the new features of the ASP components are described in more detail.

3.5.1 Network ASP

To optimally deploy a service in an active network, the network characteristics have to be considered during the deployment process. The previous work on the deployment in the FAIN project focused on element level aspects and thus extensions on the network level were required. This section describes the design and implementation of the network level ASP, which has been significantly extended since the last public description of the FAIN ASP system [4].

3.5.1.1 Network ASP Manager

As an ASP component having access to this kind of network view, the Network ASP Manager as the representative of the Network ASP is introduced. The Network ASP is the network level counterpart of the Node ASP.

The Network ASP Manager introduced in D5 [1] did not have such a network view since the ASP part of D5 focused on the Node ASP design only. Therefore the Network ASP Manager was used as a simple initiating component on network level for the deployment of a service to a single node. This Network ASP Manager had no network knowledge of any kind; it knew only about one single node where it initiated deployment of a single service.

The current design of the Network ASP includes such a network view.

This network view is restricted to service specific information, so the Network ASP has exclusive knowledge about services and contributes it at specific stages of the service deployment process to support decisions that have to be taken at network level first, then to be able to continue the service deployment process on element level.

The service information is structured according to the access provided to the Network ASP and Node ASP, as shown in Figure 66 below.

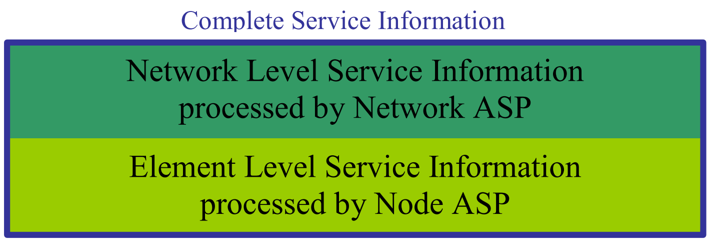

Complete Service Information
Network Level Service Information processed by Network ASP
Element Level Service Information processed by Node ASP

Figure 66. Composition of the complete service information

The Network ASP has exclusive knowledge of network level service information and the Node ASP has exclusive knowledge of element level service information in terms of exclusively processing of network level service information or element level service information, respectively.

In contrast to other interactions at specific stages of the full service deployment, i.e. knowledge sharing for requirements matching as shown in Figure 10, where exclusive knowledge of sub-systems is shared with other sub-systems, the knowledge of service information on the network level and on the element level is not shared between the Network ASP and the Node ASP.

The Network ASP does not have access to service information classified for the element level and the Node ASP does not have access to service information classified for the network level.

Since service specific information is held in service descriptors, this knowledge partitioning is realized by distributing the service specific information in network service descriptors and node service descriptors, for interest at network level and at element level respectively. The network level service descriptor is described in detail below in section 3.4.2.

The only ASP component that processes network level service descriptors is the Network ASP Manager. The Network ASP Manager is similar to the Service Creation Engine (SCE) on the element level, being exclusively responsible for parsing the element level service descriptors.

In contrast to the Network ASP Manager design described in [1], the current design presents a Network ASP Manager that acts solely on the basis of service information contained in the network level service descriptor. Since Node ASP processing is totally dependent on the service information in the element level service descriptor, current ASP system operation is completely service description based.

The architecture depicted in Figure 67 shows the main components of the Network ASP Manager. Due to the exclusive use of a Network Level Descriptor Parser the Network ASP Manager is aware of all service information included in network level service descriptors.

Processing of requests to the Network ASP Manager produces service related data that have to be stored and managed in the Service Database to be able to manage service instances and other service related data needed for internal purpose.

The Network ASP Manager has a CORBA interface for interoperability with other FAIN sub-systems. The Network ASP Manager has been implemented as an agent running on top of the Grasshopper platform. As a Grasshopper Agent the Network ASP Manager has an interface to the agency it is running on for agent related activities. For the Network ASP Manager this interface is necessary to create and send a Deployment Agent to dedicated nodes.

The Network ASP, or its representative the Network ASP Manager, is the central access point for any service related request. Therefore it provides appropriate operations, i.e. to manage service descriptors, to extract information from network level descriptors, to move code to specific nodes, to install, instantiate and configure services on specific nodes or to find out which nodes fit best to a specific service and its components respectively.

The ASP with its extension on the network level, the Network ASP, processes service information on two levels. Network level service information has the highest priority in service information processing, and has to be performed prior to the processing of element level service information.

The Network ASP is the central access point of the ASP on the network level. The ASP on the element level presents no interface to non-ASP components.

Network nodes perform network service information processing before proceeding to process element level service information.

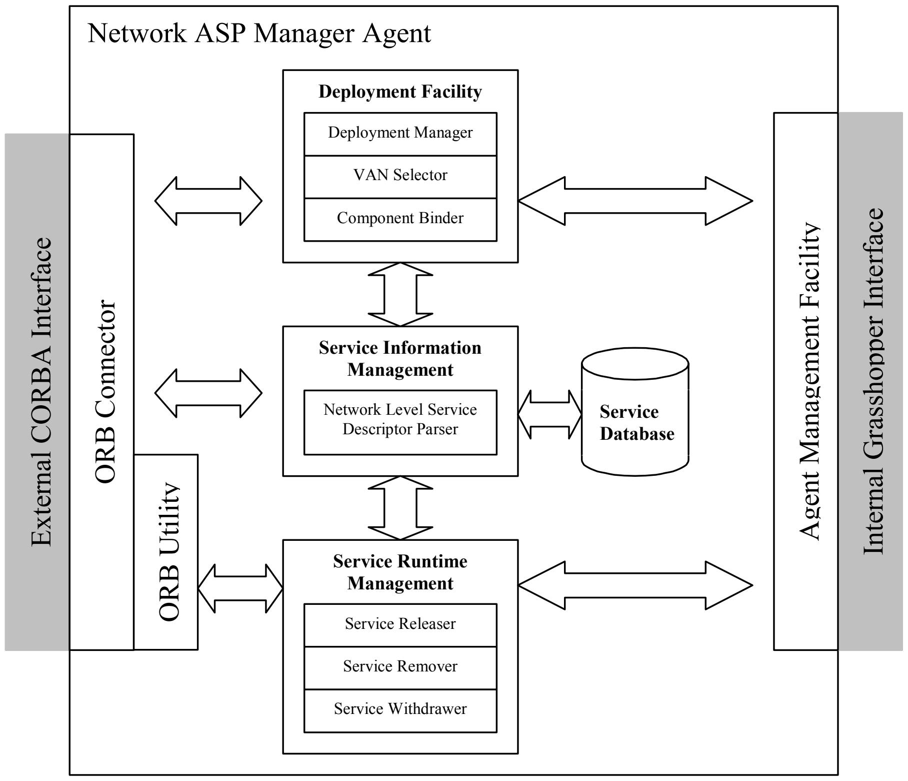


Figure 67. Network ASP Manager Architecture

3.5.1.1.1 Requirements Matching

One of the core functions of the Network ASP is to map service requirements to target environments. The determination of nodes of an active network valid for execution of specific components of a service is depicted in Figure 67.

To fulfil this task the Network ASP has knowledge of service requirements, as far a service is known to the active network, as it is stored at service release time in the Network Service Registry and the Network Service Repository. This knowledge is exclusive to the Network ASP in that other FAIN subsystems can only get this information by accessing the network ASP.

This knowledge about services includes requirements on the network level concerning service topology.

Methodology

The ASP has no knowledge of network resources or network topology, either on the network level or on the element level, although it totally relies on this information to match service requirements to target environments. Access to this information depends on non-ASP sub-systems.

The Resource Manager, as a component of the NMS, has knowledge about network resources and network topology of an active network, so it can share this knowledge with the Network ASP. The Network ASP has exclusive knowledge about service specific information.

This knowledge sharing that is necessary for requirements matching fulfilment is depicted in Figure 68.

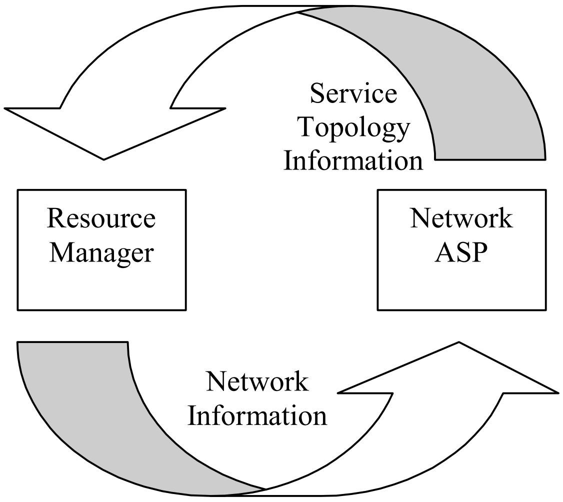


Figure 68. Knowledge sharing for Requirements Matching

If service components must be mapped to active nodes of a potential VAN, the Network ASP that is responsible for this task receives information about the network topology of the potential VAN. In addition indicators are included which classify the service topological role of each node of the potential VAN.

For this purpose a VAN consists of nodes and links connecting them. The structure of a VAN can be seen in Figure 69.

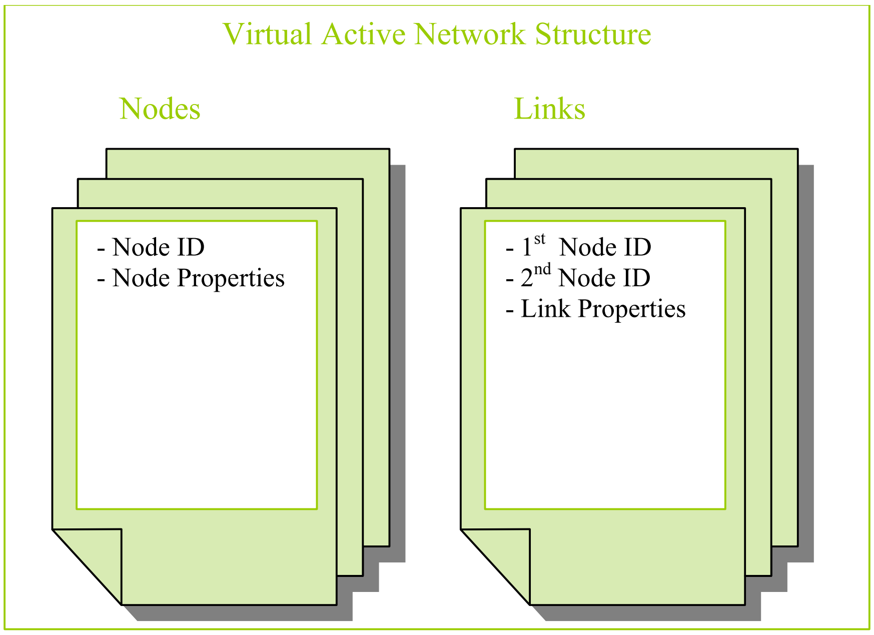


Figure 69. Virtual Active Network Structure

A node is uniquely identified by the NodeID, and node properties further characterise the node.

The following node properties are specified:

- IP address of the computer hosting the node (IPAddress)
- Classifier for the QoS level (CompQoSClass)
- Node wide unique identifier of a Virtual Environment which is unique for each Service Provider (VEID)
- Classifier for the node role (INPUT/OUTPUT)

A link connects two nodes. The specification of two NodeID's makes clear which node connections exist. This represents the network topology within the potential VAN.

The characteristics of a link are described in the link properties, as follows:

- Average bandwidth:	Value specifying the average bandwidth needed for service fulfilment
- Average throughput:	Value specifying the average throughput needed
- Link type:	Classifier for the link type
- Jitter:	The jitter parameter of the link.
- Latency:	The latency of the link.

The VAN information provides information to the Network ASP, in particular which nodes having which characteristics exist to deploy a service.

Using the service topology information from the Network ASP the RM classifies nodes within the potential VAN according to their specific roles and their ability to deploy specific service components.

Since the Network ASP has no knowledge about network resources, the RM performs this classification of nodes on network level.

The Network ASP's task now is to check if the service requirements are also met on the element level. For this purpose a so-called Deployment Agent is sent to the element level on each node to check these node wide service requirements. At this time the Node ASP is taking part in this investigation process as it, exclusively, has the functionality to check if node wide service requirements can be met by the target environment.

The component of the Node ASP used to do this check of node wide service requirements is the Service Creation Engine (SCE) as depicted in step 6 in Figure 69.

The SCE parses the node service descriptor and resolves any dependencies. This means determining the dependencies between services and between service components and the target environment in which it must execute.

If the dependency resolution finds out that the target environment is valid for the service to be deployed, the Network ASP assigns the appropriate node of the potential VAN to the service or its component respectively.

This process of checking node wide service requirements on the element level is done until every service component has been assigned to a node of the potential VAN or no other nodes in the potential VAIN are left to assign to the remaining service components. . If none of the proposed VAN's is suitable for all components of a service a failure report is given back to the requester the NMS.

The Network ASP adds a classifier to each node of the chosen potential VAN to identify nodes that will be used for the deployment of a service or its component respectively. The only information about specific components that will be applied on these nodes is the kind of Execution Environment (EE) that needed.

The NMS, which receives this modified potential VAN, recognizes which nodes will be involved in the service deployment due to the classifier given by the Network ASP. Therefore the NMS can reserve resources and create resources prior to service deployment. Resources to be created could be Virtual Environments (VE's) and Execution Environments within the VE's, in which to apply the service or its components respectively.

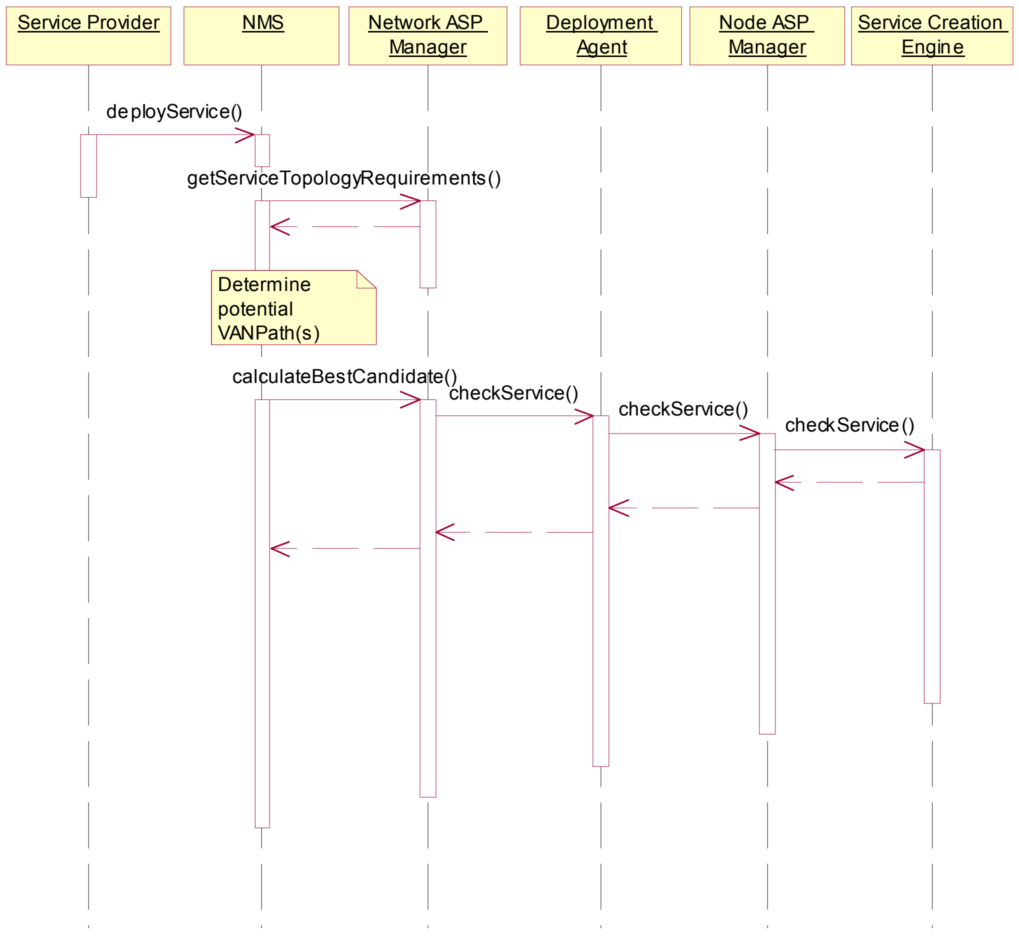


Figure 70. Requirements matching Sequence Diagram

3.5.1.1.2 Service Deployment

Once the Network ASP Manager has determined the nodes of the potential VAN that will be used for service deployment and the NMS has reserved and created the required resources, the service deployment process can be initiated.

Within the NMS, service deployment is initiated by the Node ASP Manager at the request of the Network ASP Manager.

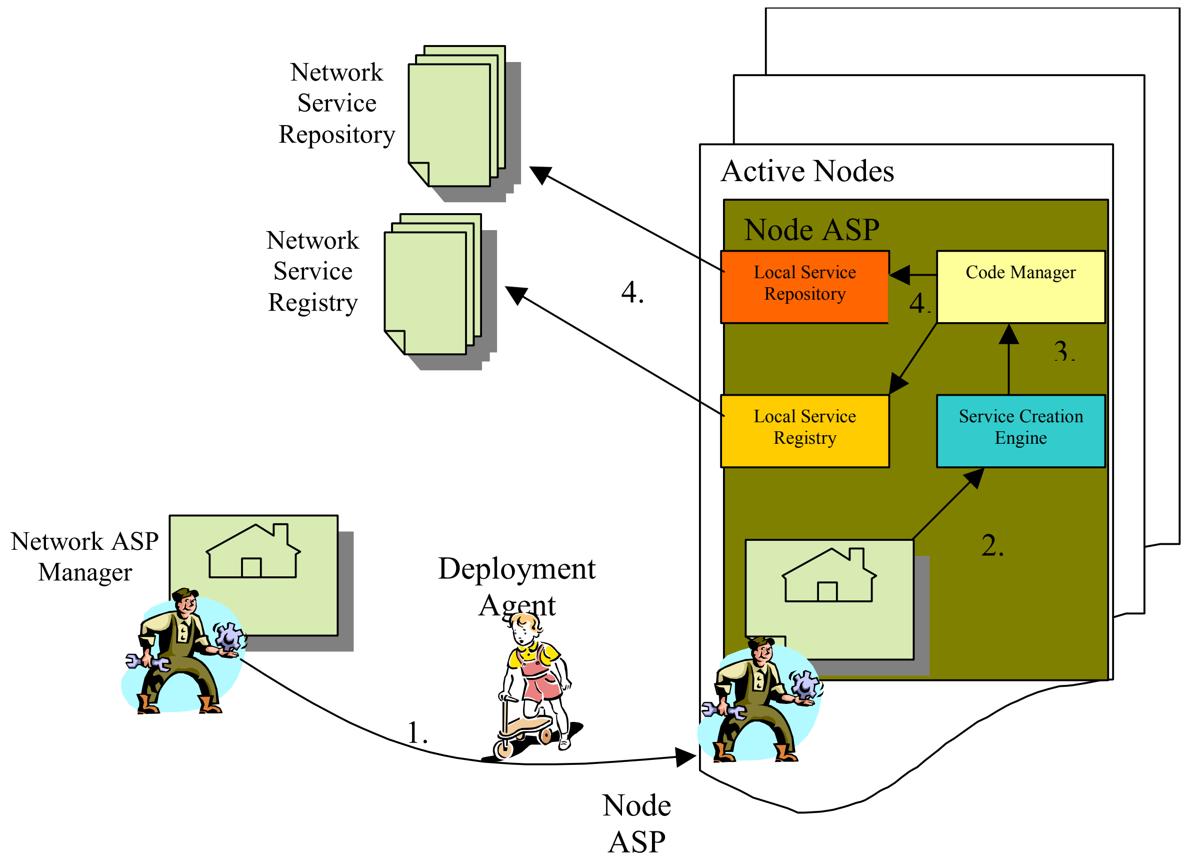


Figure 71. Service Deployment

Service Distribution

The Network ASP Manager contacts the Node ASP Manager of each involved active node to start the service deployment on these nodes. The node ASP Manager is a static agent located at the Node ASP's agency in each active node. The Deployment Agent works as a carrier for this task information (step 1).

This task is passed by the Node ASP Manager to the Service Creation Engine (SCE) for node-wide service deployment (step 2).

The SCE parses this node-wide service descriptor. The parsing results are transmitted to the Code Manager as a list of software components, and their associated service descriptors, that need to be fetched. Furthermore dependent service components as mentioned in the node-wide service descriptor are also passed to the Code Manager (step 3).

The required service components are fetched by the Local Service Repository and the Local Service Registry or its network counterparts respectively on request of the Code Manager (step 4).

Service Installation and Configuration

The Code Manager asks the Node Manager for known resources, such as available EE's and where to pass the EE-specific part of the node-specific service descriptor for service code module installation and instantiation.

The Code Manager initiates the EE-specific installation and service-specific configuration on the specific EE by using a defined EE-specific interface. The service is configured according to the service properties given to the Code Manager at instantiation request time.

The resulting service reference is passed to the SCE.

The SCE passes this service reference to the Node ASP Manager which passes it back to the Network ASP Manager using the Deployment Agent.

The Network ASP Manager passes this service reference to the requester NMS.

3.5.1.1.3 Service Reconfiguration

The steps of service deployment are combined as step 1 in Figure 72. After a service has been deployed successfully, the runtime environment could change in a way that it no longer fits the requirements of the deployed service and its components.

Reconfiguration can take place in case of an inconsistency of requirements of a service already running on a node and the target environment of this node. If such an inconsistency is identified a new matching of service requirements to target environments is initiated automatically. This kind of reconfiguration, called dynamic reconfiguration is initiated within the ASP.

Reconfiguration is also possible on request by a Non-ASP management components, like the NMS, to manage service components on specific nodes to meet different target or service environments.

So we differentiate between two cases of reconfiguration, the dynamic reconfiguration during service runtime and the management controlled reconfiguration as described in the following sections.

Dynamic Reconfiguration

To be able to react automatically to a change of node resources, resource monitoring has to take place. The Node Manager of the Node Framework on every node offers two different kinds of resource monitoring method. The first monitoring method offers a callback interface that notifies the registrar of a callback interface for a specific resource if a specific threshold is reached. The second method offers monitoring by polling.

An ASP component, the Reconfiguration Manager, is involved in this resource monitoring. This monitoring component is located within the Node ASP so it acts solely on the element level. The Reconfiguration Manager steps on stage right after instantiation of a service which means also the use of specific resources by this service.

The Code Manager as the Node ASP component responsible for initiating installation and instantiation of services orders the Reconfiguration Manager after instantiation of a service to observe resources used by this service as step 2 of Figure 72 illustrates. These resources and their specific values for a service make up the basic conditions needed for a service. If these crucial resources and their values do no longer meet the basic conditions of a service, a reconfiguration of this service must take place. The Code Manager specifies exactly which resource to monitor to meet which values.

The Reconfiguration Manager registers a callback interface (resource observer) at the Node Manager for every resource (i.e. of services and sub-services) it needs to monitor as depicted in Figure 72 as step 3.

In the case of a specific threshold being reached, the Node Manager of the Node Framework sends an event to the Reconfiguration Manager, also shown in figure 8 as step 3. The Reconfiguration Manager then informs the Network ASP Manager as step 4 about a non-valid target environment for the service it is responsible for.

The Network ASP Manager now has to inform the Network Management System (NMS) about the need to search for a different node for a service (service reconfiguration) as it does in step 5, since the ASP is not allowed to decide about a re-matching of service requirements to target environments on its own.

A Component of the NMS then decides whether a new matching of service requirements to target environments should take place for a service whose runtime environment does not match its requirements.

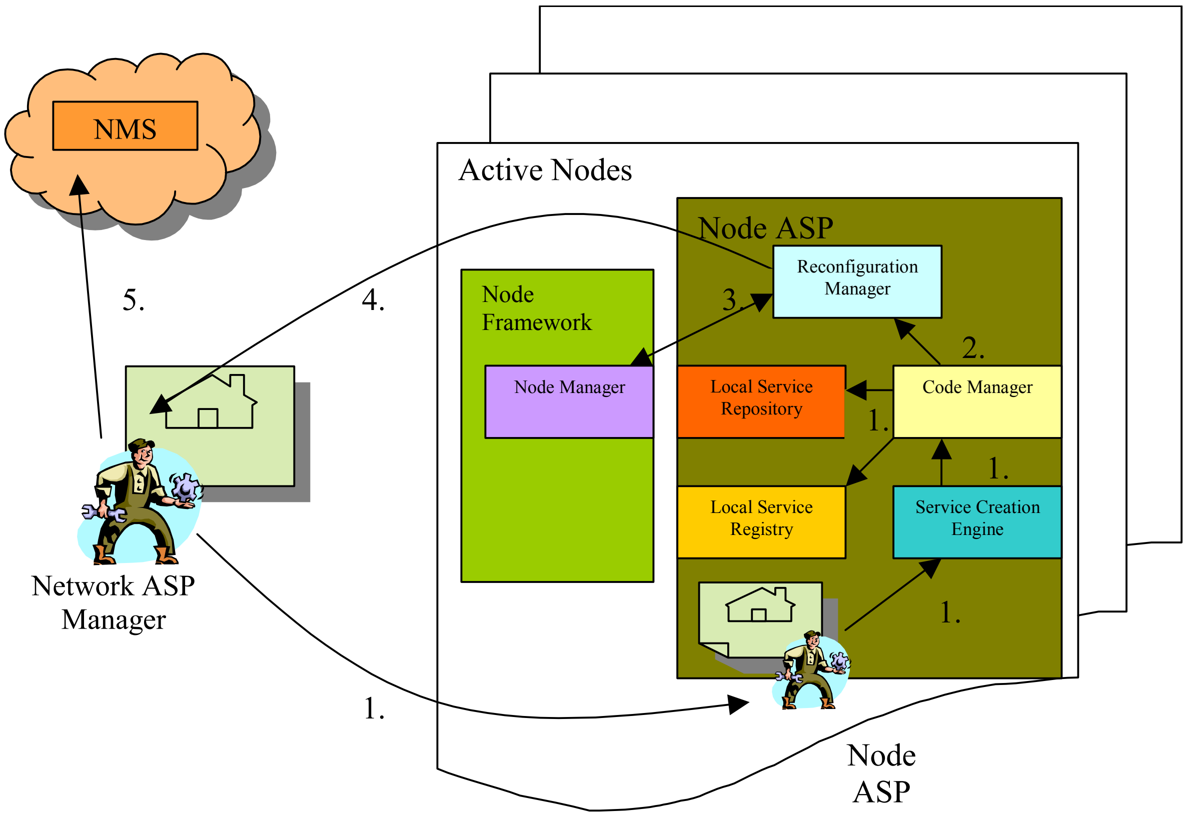


Figure 72. Dynamic Reconfiguration, 1st level

Within the dynamic reconfiguration we differentiate between two levels.

The 1st level is reached when the result of the new matching of service requirements to target environments is actually the same node the service is already running on. In this case this service needs to be reconfigured in terms of a new determination of service implementations by the SCE for the changed target environment. The steps performed are the same as they are for the first deployment of a service as depicted in Figure 72.

The 2nd level is reached when the new matching of the target environment of the node a service is actually running on and the requirements of this service results in a different node from the node the service is running on. In this case the service has to be started on the new node. The node that is no longer useable is called 'non-valid node'. The node that was found after a new requirements matching to target environments is called 'valid node'.

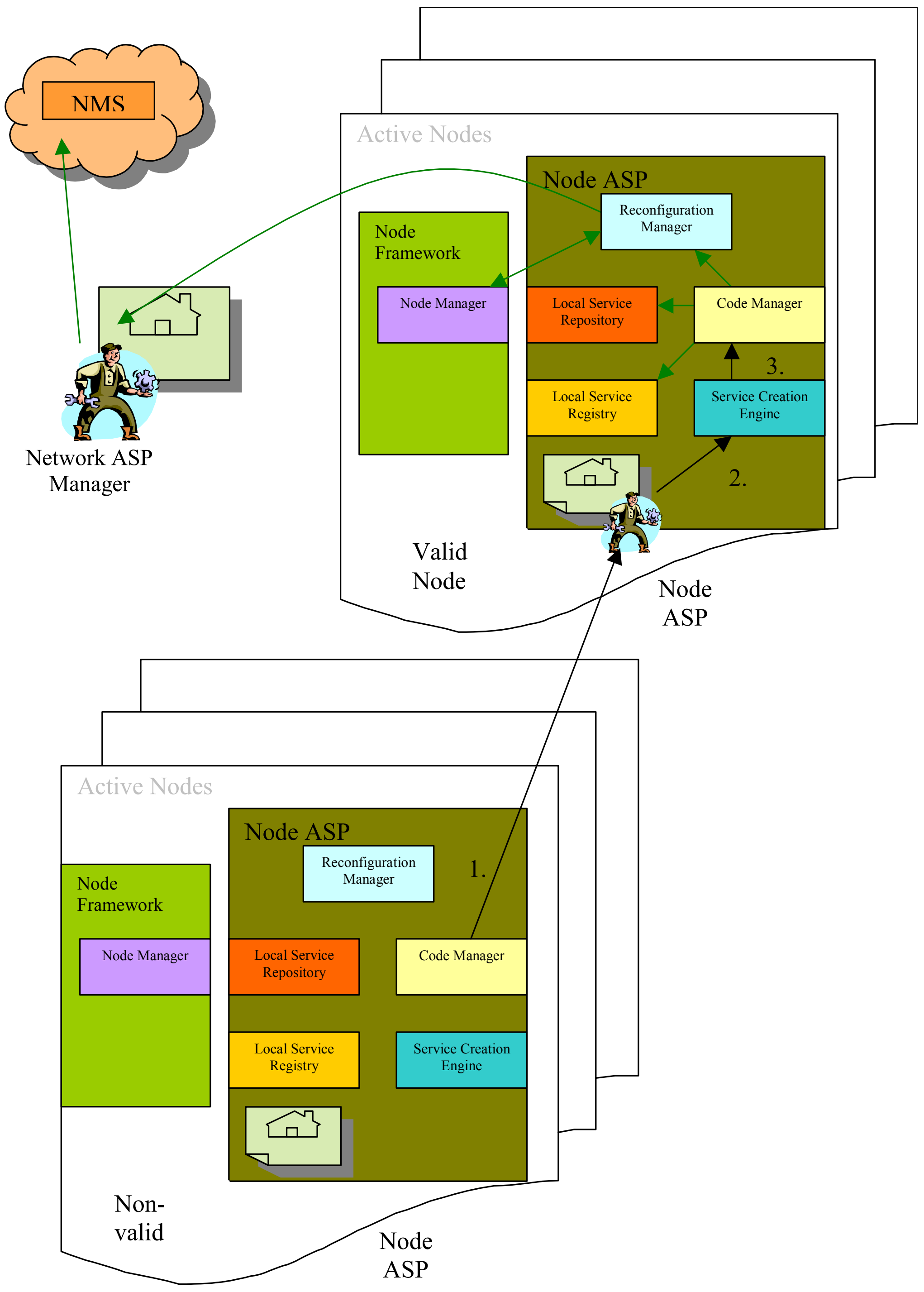


Figure 73. Dynamic Reconfiguration, 2nd level

A dynamic reconfiguration at the 2nd level must make sure that the properties and status of the service that was already running on a valid node are preserved, so the service can continue under the same service related conditions as before. This service status preservation is done by the Code Manager of the non-valid node at the request of the Node ASP Manager of the valid node.

The Node ASP Manager of the valid node is contacted by the Code Manager of the non-valid node receiving the preserved service properties as depicted as step 1 in Figure 73 which are passed to the Code Manager of the valid node at instantiation request time over the SCE as shown in steps 2 and 3 that is responsible to parse the element level service descriptor and resolve dependencies within a service.

Management controlled Reconfiguration

In contrast to dynamic reconfiguration where the Reconfiguration Manager alerts the need for a service reconfiguration, controlled reconfiguration is requested from outside the ASP.

The Node ASP has no logic or information upon which to base a decision to initiate a management controlled reconfiguration. Instead the decision to reconfigure a service is computed by the NMS or the EMS, so this kind of reconfiguration is only done by the ASP on request.

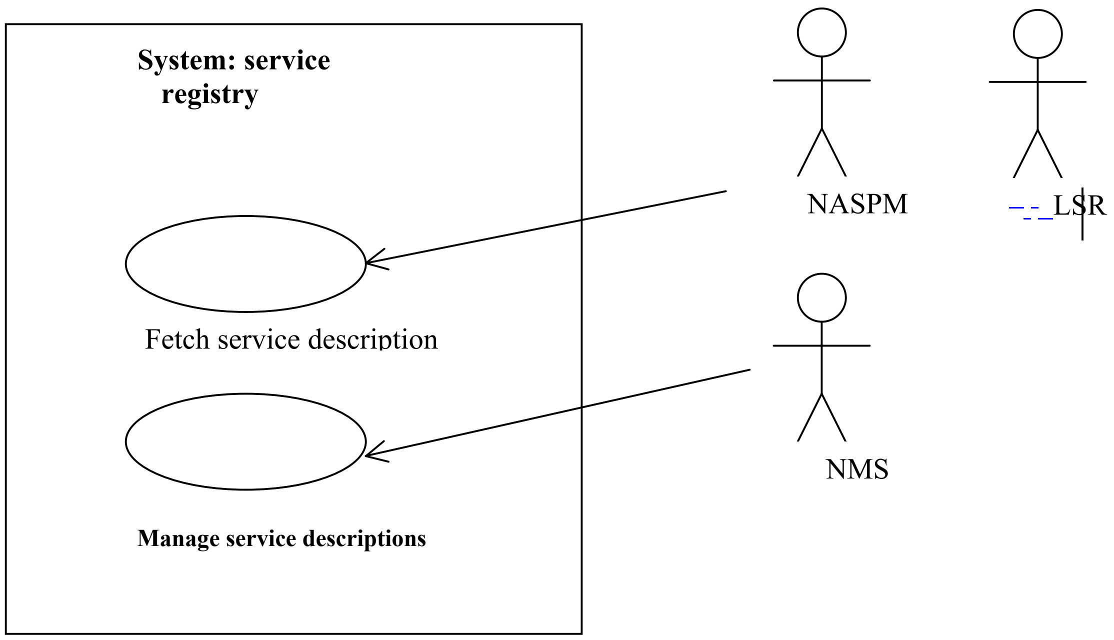
Management controlled reconfiguration can be initiated in cases where the port bindings between service components have to be changed, for example a new service component could be added to a node to fulfil a task in cooperation with an already running service component, in which case the ports of these two service components are bound.

3.5.1.2 Service Registry

In the ASP part of the FAIN architecture, the service registry is responsible for managing the description of services that can be loaded into active nodes (register, unregister, find services).

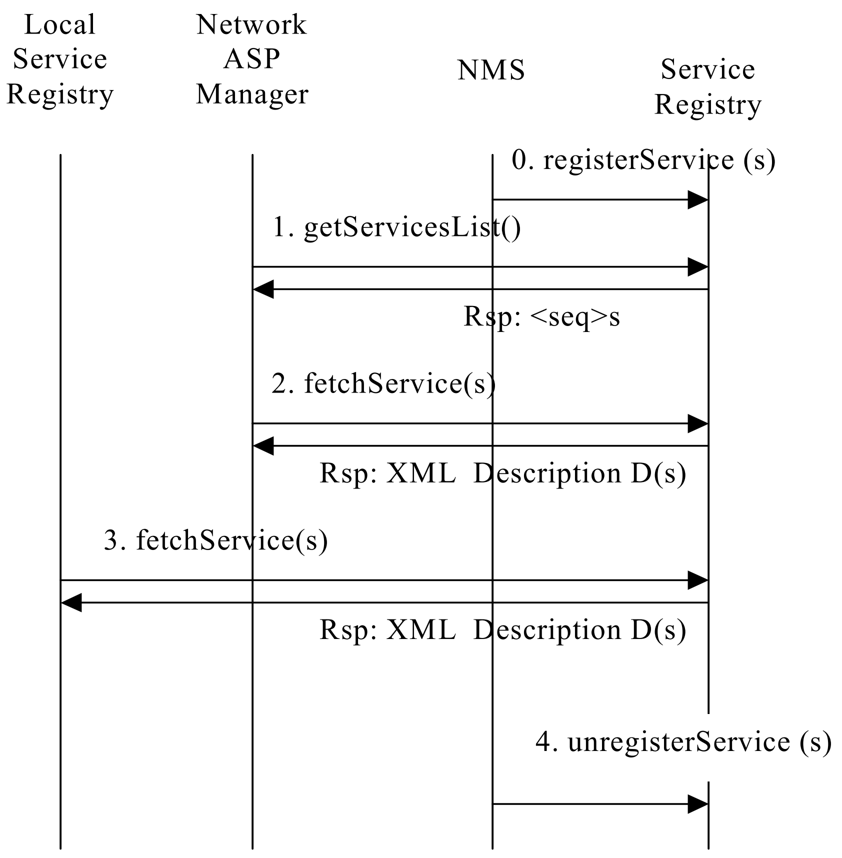
This section aims to briefly present the design of this component and to show its interfaces (defined in IDL).

3.5.1.2.1 Use cases



The actors are: the NASPM and the NMS.

- NASPM: Network Active Service Provisioning Manager.
- LSR: Local Service Registry
- NMS: Network Management System.



Use case 1.

The first use case starts when the NASPM needs to fetch the description of a service. First the NASM gets the list of the available service descriptions by calling the method *getServicesList* of the CORBA interface. Then the NASPM can choose a service in the list and it requests its descriptions (a sequence of XML file) by calling the method *fetchService*. There might be several descriptions for a service, in which case several XML scheme might be returned.

Use case 2.

The second use case starts when the NMS wants to install a new service in the active node. The NMS registers a new service description with *registerService* and unregisters it with *unregisterService*.

Use case 3.

The third use case starts when the LSR is asked by the Node Code Manager for the description of a service it doesn't have locally. It then asks for the descriptions of this service from the service Registry by calling the method *fetchService*.

3.5.1.2.2 Design

The service registry sends information to the ASP Network Manager based on the ASP Network Manager request. The latter is responsible for combining this information to fulfil the user request: no further processing is needed in the case of (un)register requests, but there is more work to do for a deployment of the user's service (resolving service dependencies and asking the service registry other information, asking the download of the right code at the right code repository, downloading the code to the appropriate active node…).

The service registry must register new services, unregister old services, find services (based on the name).

The service name must be unique (in order to clearly distinguish services): it is composed by the name of the service concatenated with the name of the service provider (example: VideoTranscoder_FTR&D).

No public attribute is necessary. Only 4 methods are public

- registerService: In order to register a service into the Service Registry, the service name must be passed with a descriptor, describing the service. This descriptor is a XML file, mapping the Chameleon requirements. If the service is already registered (one previous version has already been registered) then the service registry registers this new request as a new version of the service and increments the number of the version.
- unregisterService: Only the service name is passed to the service Registry, and the latter removes it from the database and removes all the descriptions related to it.
- fetchService: Only the service name is passed to the service Registry, and the latter is responsible for retrieving the XML descriptors in the database and sending it back to the client (ASP Network Manager or Local Service Registry). If there are several versions of the service (then several XML descriptors), all the XML descriptors are sent back.
- getServicesList: No input parameter is given. When receiving this request, the Service Registry sends back all the registered services.

Some exceptions are also defined: checking the correctness of the name, the syntax of the XML descriptor, etc.

In the initial implementation, it is planned to store this registry in regular files and not to use a database yet (because the data size will not be large).

It has been decided to pass the XML file to/from the Service Registry as a String. It is assumed that a String can be long enough to represent the XML descriptor.

Another option was to implement a HTTP server and to request a GET/PUT method on this server, but in this case we are no longer in a CORBA architecture (as it has been defined by FAIN members).

3.5.1.2.3 IDL Interface

```
typedef string ServiceComponentID;
typedef string ServiceName;
typedef sequence<ServiceName> ServiceNames;
typedef string ServiceComponentDescriptorRef;
typedef sequence<ServiceComponentDescriptorRef> ServiceComponentDescriptorRefs;

exception serviceNotFound{
string name;
};

exception invalidServiceName {
string name;
};

exception invalidXMLDescriptor {
string serviceName;
};

interface ServiceRegistry {

void registerService(in ServiceName serviceToRegister,
            in ServiceComponentDescriptorRef XMLDescriptor)
raises (invalidServiceName
  , invalidXMLDescriptor);

ServiceComponentDescriptorRefs fetchService(in ServiceComponentID serviceToFecth)
raises (invalidServiceName, serviceNotFound);

void unregisterService(in ServiceName serviceName)
raises (invalidServiceName, serviceNotFound);

ServiceNames getServicesList();

};
```

3.5.1.3 Service Repository

The service repository contains the implementation components for the services which are available in the network. These components can be specific to an implementation from a particular vendor, or for a specific EE-type. The repository stores only the code files. Additional information, about which components are required for the service, or how these must be configured, is stored in the service registry. In the FAIN Network architecture, there is a central Service Repository for each administrative domain.

The main requirement of the service repository is to function as a file-server, which contains the code modules that can be injected into the active nodes. The repository can be implemented using existing file transfer mechanisms. In the initial implementation an http server is used as the service repository. An alternate implementation using plain sockets is also under development.

3.5.1.3.1 Service Repository: Interfaces, implemented operations

The Service Repository offers the following operations:

- `storeComponent`
- `deleteComponent`

These two operations are used to add/remove software components to/from the repository.

- `getComponent`

  This operation is used to download a specific component to the active node. The component is identified by its name, which must be unique.

The main Service Repository operation required for the initial implementation is getComponent. As the repository is realized using an http server, this operation is actually implemented by http. The components are stored in the server in a directory structure. Files are stored in a path of the form /EEType/Service/developer/component filename. All necessary information for the creation of this path is included in the naming scheme used for the components, so by knowing the component's name it is easy to construct the path to its location.

At the moment the operation required for inserting new implementation components has not been fully implemented.

3.5.2 Node ASP

On the node level, the following components make up the ASP system, as shown in the node ASP block in Figure 74, Node ASP manager, Service Creation Engine and Code Manager.

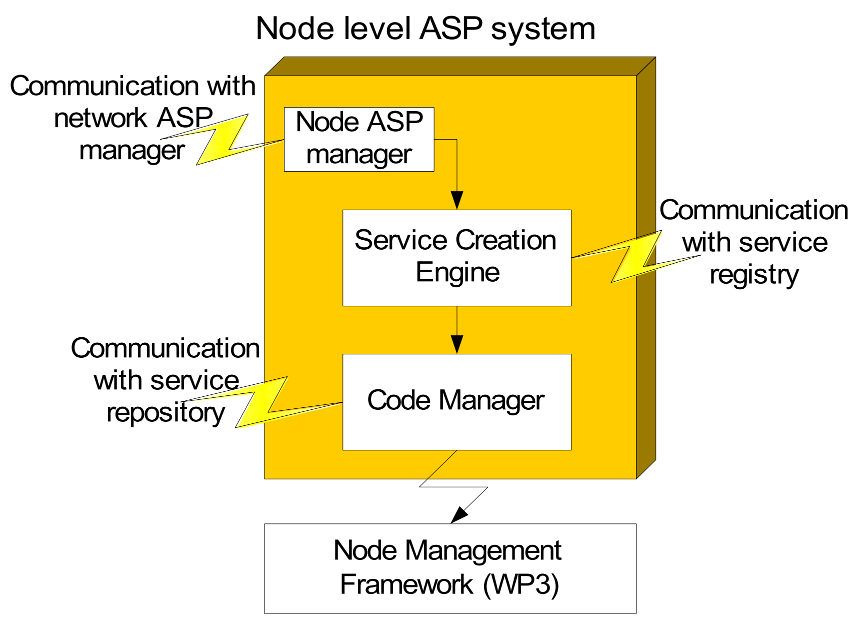

Node level ASP system
Communication with network ASP manager
Node ASP manager
Service Creation Engine
Communication with service registry
Communication with service repository
Code Manager
Node Management Framework (WP3)

Figure 74. Node level ASP

The **Node ASP Manager** is the equivalent of the network ASP manager, but on the node level. The network ASP manager communicates with the node ASP manager in order to request the deployment, upgrading and removal of service components.

The **Service Creation Engine (SCE)** selects appropriate code modules to be installed on the node in order to perform the requested service functionality. The service creation engine matches service component requirements against node capabilities and performs the necessary dependency resolution. Since the service creation engine is implemented on each active node, active node manufacturers are enabled to optimise the mapping process for their particular node. In this way it is possible to exploit proprietary, advanced features of an active node. The selection of service components is based on service descriptors. Moreover, service descriptors describe how service components are bound to each other. This type of information is extracted by the SCE and passed to the code manager.

The **Code Manager** performs the execution environment independent part of service component management. During the deployment phase, it fetches code modules identified by the service tree from the service repository. It also communicates with Node Management to perform EE-specific part of installation and instantiation of code modules. The Code Manager maintains a database containing information about installed code modules and their association with service components.

The **Local Service Registry** (LSR), its counterpart of the Service Registry in the active node and acts as a cache-enabled client of the Service Registry in the node level. If the local service registry does not have the description of the given service, it asks the network Service Registry for it. Otherwise it returns a local copy of the node level service descriptor.

The **Local Service Repository** is a client of the Service Repository and fetches code modules on demand.

3.5.2.1 Node ASP Manager

The Node ASP Manager is implemented as a stationary agent running in a Grasshopper agent system. The connection of the Node ASP Manager to the Demux component is realized by using the Communication Service facilities of Grasshopper. A (mobile) deployment agent will be received via that Communication Service which is encoded in an ANEP packet and received from network level (i.e. Network ASP manager).

3.5.2.1.1 Node ASP Manager: Interfaces, implemented operations

There is an inter-agent system interface for exchange/migration of the mobile deployment agent provided. This interface is integral part of the Grasshopper environment distribution and does not need to be described here. For further description, please refer to the Grasshopper documentation, which can be found at www.grasshopper.de.

3.5.2.1.2 Node ASP Manager Activity diagram

This is provided in the section above, with the Network ASP Manager.

3.5.2.2 Code Manager

The **Code Manager** performs the EE independent aspects of service component management. During the deployment phase, it fetches code modules identified by the service tree from the service repository. It also communicates with Node Management to perform EE-specific part of installation and instantiation of code modules. The Code Manager maintains a database containing information about installed code modules and their association with service components.

3.5.2.2.1 Design

The Code Manager is a node-level ASP component, which maintains the information about the code modules, **installed** on the node. This component is contacted after the Service Creation Engine has resolved the dependencies of the service component requested to be deployed.

The service component information comprises:

- Service Component dependencies:
    - o Resolved inter-component dependencies, in the form of a list of all the service components that the given service component depends on.
    - o Environment dependencies, i.e. the dependencies on the execution environment that the code module associated to a service component is supposed to run in.
- Service Component local installations:
    - o Expiration date,
    - o VE identifier and EE identifier, where the code modules are installed.

The Code Manager holds the information about the installed service components in a data structure forming a directed acyclic graph (DAG). The nodes in the data structure represent the service components installed on the node, whereas the edges represent the dependencies between these components. The data structure used to keep this information is depicted in Figure 70. The DAG has two levels: the first level consists of nodes representing service components that were requested to install by the SCE. Nodes on the second level represent the service components that the service components from the first level directly or indirectly depend on.

The information maintained by the Code Manager is updated by:

- SCE in case it requests fetching and installing a service component
- Code Manager itself whenever a service component expires and needs to be uninstalled from a given target environment.

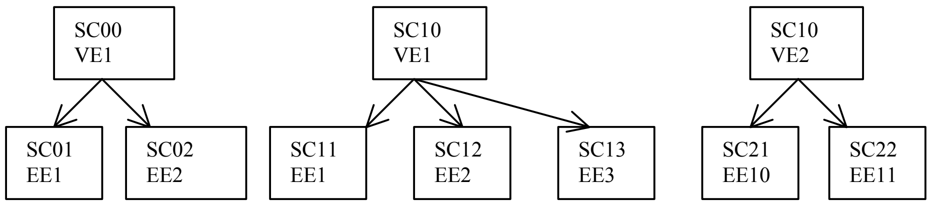


Figure 75. Data structure with the code module information in the Code Manager.

IDL Specification

The Code Manager-related interfaces and data structures are specified in IDL. They defined in the ASP module and presented in Listing 1.

InstallationTarget

The data structure represents a target environment where a code module is installed. The environment is identified by a pair of the Virtual Environment Identifier and Execution Environment Identifier.

ServiceComponentInfo

The data structure represents some information associated with the service component from which the code module is referenced. The information contains the code module identifier, the code module identifier of the parent service component and the component module expiration date.

EndPoint

This data structure describes an end point of a connection between two components. It includes a reference to a service component and the corresponding port of this component.

ConnectionInfo

This data structure includes all the information needed to determine a connection between two service components. The information contains two end points of the connection as described above.

ServiceInfo

The data structure describes all the information needed to properly instantiate a service. The information includes references to the service components and a set of connections to be built up between these service components.

Code Manager Interface

This interface is provided by the Code Manager component. It is mainly used by the service creation engine (SCE), which resolves the service component dependencies and requests their fetching and installation. The interface defines three operations described below:

- `fetchAndInstallService`. The operation triggers and coordinates:
    - fetching all the code modules that a given service component depends on
    - installing these code modules into their target environments.

The input parameter the service component information including the component identifier, the VE identifying the target environment and a list of dependent service components.

- `uninstallService`. The operation triggers uninstalling the given service component and all its dependent service components from a given execution environment instance.
- `getDeploymentDescriptors`. The operation returns a list of available deployment descriptors representing possible realizations of the given service. `ServiceComponentIDNotFound` is thrown if the service component cannot be found in Service Repository.

Listing 1 Code Manager and related IDL specifcation

```
module asp {
 typedef string ServiceComponentID;
 typedef string CodeModuleID;
 typedef string ServiceName;
 typedef string VeID;
 typedef string EeID;
 typedef string CodeModuleRef;
 typedef string DeploymentDescriptorRef;
 typedef sequence<DeploymentDescriptorRef> DeploymentDescriptorRefs;
 typedef string Date;

 struct InstallationTarget {
 VeID veID;
 EeID eeID;
 };

 struct Property {
  string key;
  any value;
 };

 typedef sequence<Property> Properties;
 typedef Properties Configuration;




 /**
 * this type describes the metadata of an implementation (i.e. a service
 * component with a reference to a code module).
 */

 valuetype ServiceComponentInfo {

 /* the id of the service component */
 public ServiceComponentID theComponent;

 /** the Execution Environment type in which service component may run */
 public EeID target_ee;

 /** configuration information */
 public Configuration config;

 /** reference to the code module */
 public CodeModuleRef codeModule;

 /** helper function to create a valuetype */
 factory init(in ServiceComponentID theComponent,
     in EeID target_ee,
     in Configuration config);
 };

 typedef sequence<ServiceComponentInfo> ServiceComponents;


 struct EndPoint {
       ServiceComponentID componentID;
       PortName       p_name;
 };
```

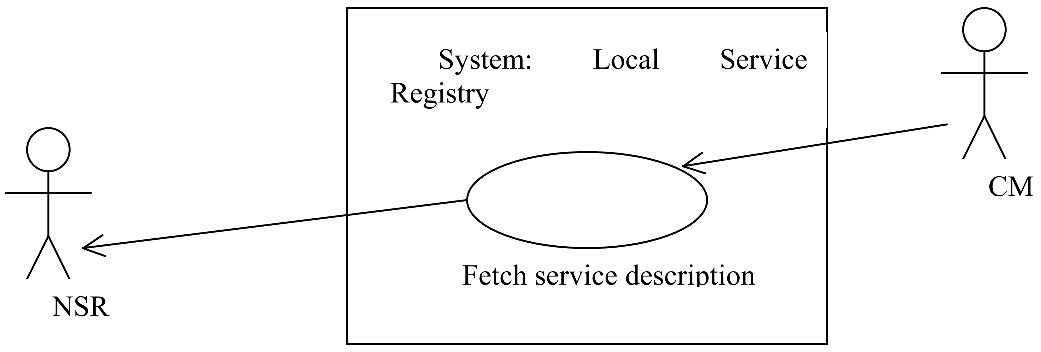
```
struct ConnectionInfo {
      EndPoint firstEndPoint;
      EndPoint secondEndPoint;
};

    typedef sequence<ConnectionInfo> ConnectionInfos;

struct ServiceInfo {
      ScInfos scInfos;
      ConnectionInfos connectionInfos;
};



exception ServiceComponentIDNotFound { string ServiceComponentID; };
exception CodeModuleIDNotFound {};
exception InstallationTargetNotFound { };
exception InstallationFailed { string reason; };

interface CodeManager {

/**
* updates the CM data base by adding information representing the
* service to deploy and all the dependent implementations and Mr.Xs
* @param service_id the service component to be installed
* @param target_ve the Virtual Environment where the service component
* is to be installed
* @param expiration_date the date until the service may be used
* @param dependent_components the resolved dependencies of the service
* component
* @exception ServiceComponentIDNotFound if one of the dependent service
* components cannot be found in the service repository
* @exception InstallationFailed if the installation process cannot be
* succeed for some other reason
*/
void fetchAndInstallService(
                  in ServiceComponentID service_id,
                  in VeID target_ve,
                  in Date expiration_date,
                  in ServiceComponents dependent_components)
raises (ServiceComponentIDNotFound, InstallationFailed);



/**
* removes all the code modules installed for a given Service Component
* @param service_id the id of the service component to be uninstalled
* @param target_ve the Virtual Environment from which the service
 component is to be uninstalled
* @exception ServiceComponentIDNotFound if a service cannot be found
* among the installed ones
* @exception InstallationTargetNotFound if the given VE is not valid
*/
void uninstallService(
            in ServiceComponentID serviceID,
            in VeID target_ve)
raises (ServiceComponentIDNotFound, InstallationTargetNotFound);


/**
* returns a list of available deployment descriptors representing
* possible Service Component IDs (i.e. various realizations of the
* given Service Component.
* @param service the service component of which available realizations
*  are to be returned
* @exception ServiceComponentIDNotFound is thrown if the service
  component cannot be found in Service Repository
*/
DeploymentDescriptorRefs getDeploymentDescriptor(
 in ServiceName service)
raises (ServiceComponentIDNotFound);
};
};
```

3.5.2.3 Local Service registry

Inside an active node, the local service registry is responsible for managing the description of services that are requested by the Code Manager and can be loaded into active nodes. If the local service registry does not have the description of the given service, it asks the network service registry for it.

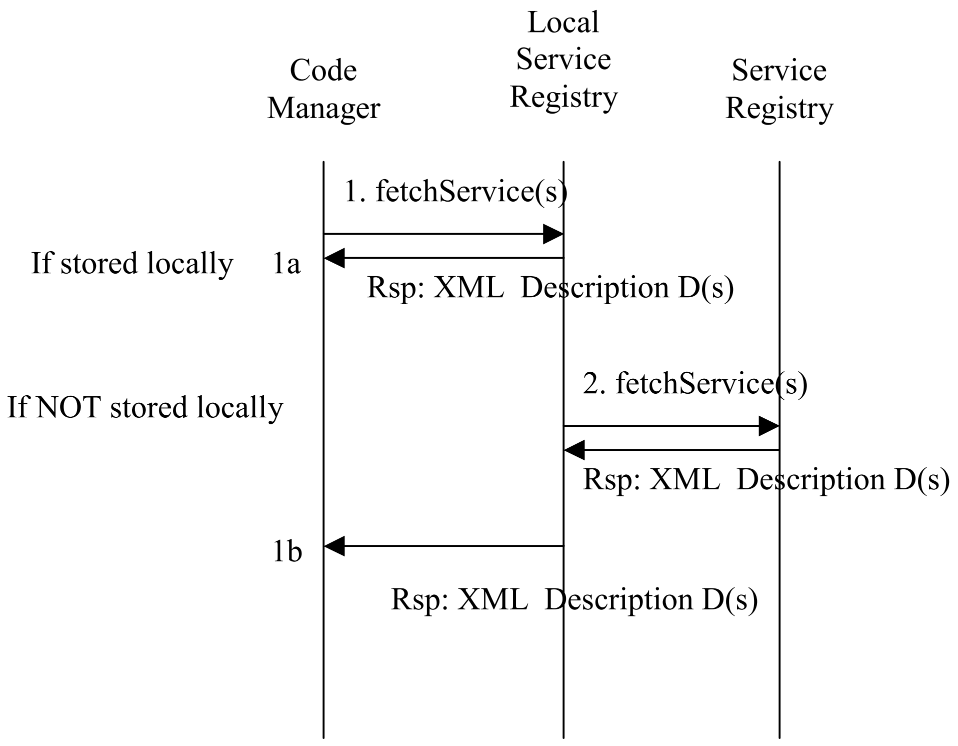
This section aims to quickly present the design of this component and to show its interfaces (defined in IDL).

3.5.2.3.1 Use cases



The actors are: the CM and the NSR.

- CM: Code Manager.
- NSR: Network Service Registry.



Use case 1.

The first use case starts when the CM wants the description of a service *fetchService*. The CM asks it to the LSR. If the latter has this description locally (in cache), it returns it to the CM. If the LSR doesn't have the descriptions of the service, it then asks it to the network Service Registry, gives it back, stores it and sends it back to the CM.

3.5.2.3.2 Design

The Local Service Registry is responsible for managing service descriptions locally inside the active node. Its role is then to fetch service descriptions and to store them (cache).

When the Code manager wants to deploy a service, it asks the LSR the descriptions of this service.

If this service has already been deployed (or requested by the CM), the LSR has keep the descriptions in cache and then can give them back to the CM.

If this description is not know locally by the LSR, then the latter will contact the network service registry and fetch the descriptions for this service.

The LSR will then store it locally (keep it in cache) and will send back this information to the CM.

The LSR can also reply to the CM if the CM wants to get the list of all available services. This option will certainly not be used because it is not the role of the CM but it is possible.

If the CM requests that, then the LSR will contact the Network Service Registry to retrieve the list of services. This list is not cached in order to get always the up-to-date list.

No public attribute is necessary. Only 2 methods are public

- fetchService: Only the service name is passed to the Local Service Registry, and the latter is responsible for retrieving the XML descriptors (locally if stored or remotely with CORBA invocation to the network service registry if not) and sending it back to the Code Manager. If there are several versions of the service (then several XML descriptors), all the XML descriptors are returned.
- getServicesList: No input parameter is given. When receiving this request, the Local Service Registry requests the network service registry to get this information and sends back all the registered services (received from the SR) to the CM.

Some exceptions are also defined: checking the correctness of the name, the syntax of the XML descriptor, etc.

3.5.2.3.3 IDL Interface

The IDL definition of the Local Service Registry is included below.

```
typedef string ServiceComponentID;
typedef string ServiceName;
typedef sequence<ServiceName> ServiceNames;
typedef string ServiceComponentDescriptorRef;
typedef sequence<ServiceComponentDescriptorRef> ServiceComponentDescriptorRefs;

exception serviceNotFound{
string name;
};

exception invalidServiceName {
string name;
};

exception invalidXMLDescriptor {
string serviceName;
};
```

```
interface ServiceRegistry {

ServiceComponentDescriptorRefs fetchService(in ServiceComponentID serviceToFecth)
raises (invalidServiceName, serviceNotFound);

ServiceNames getServicesList ();

};
```

3.5.2.4 Local Service Repository

The Local Service Repository caches the service implementation components that have been recently downloaded to the active node, so that if they are requested in the future it will not be necessary to retrieve them from the network.

The number of components that are cached depends on the available storage space on the node. If available space is exhausted, a replacement algorithm is applied, to delete an unnecessary component from the cache in order to store a new one. The repository itself does not have the necessary logic for these checks. This is the responsibility of the code manager. In this context the local cache can be considered as a simple “back-end” of the code manager.

3.5.2.4.1 Local Service Repository: Interfaces, implemented operations

The interface of the Local Service Repository has been specified in IDL. The implementation is in pure Java and so the use of CORBA objects has not been necessary.

The IDL interface of the Local Service Repository is presented below.

Listening: IDL for Local Service Repository.

```
typedef string CodeModuleID;
typedef string CodeModuleRef;

 exception BadComponentName { string componentName; };

 exception ComponentNotFound { string componentName; };

 interface ServiceRepository {

 boolean storeComponent(in CodeModuleRef codeComponent,
    in CodeModuleID componentName)
 raises (BadComponentName);

 boolean deleteComponent(in CodeModuleID componentName)
 raises (ComponentNotFound);

 CodeModuleRef getComponent(in CodeModuleID componentName)
 raises (ComponentNotFound, BadComponentName);

 };
```

All of the three operations of the Local Service Repository interface have been implemented for the milestone, although only one of them, *getComponent*, will be essential for the demonstration.

- `getComponent`

  This operation is responsible for retrieving a code module. The repository first checks if the requested component is cached locally. A hash table is used as an index for the cached components. If the code does not exist locally, it is downloaded from the network service repository. In this implementation, where the network service repository is an http server, the download of code to the node is done using http. The operation returns a reference of the local file, which contains the code.

- `storeComponent`

This operation is used to store a new component in the cache.

- `deleteComponent`

This operation deletes a cached component. It is used by the code manager, when it decides that a stored component must be deleted, either because it has been cached for a long time or has to be refreshed, or because more disk space is required to cache other components.

There are also the following two exceptions defined:

- `BadComponentName` is thrown when the component name given as input is not valid, so the Local Service Repository is not able to request the file from the code server.
- `ComponentNotFound` is thrown when the requested component could not be located in the local cache or retrieved from the network service repository.

3.5.2.4.2 Local Service Repository: Activity diagram

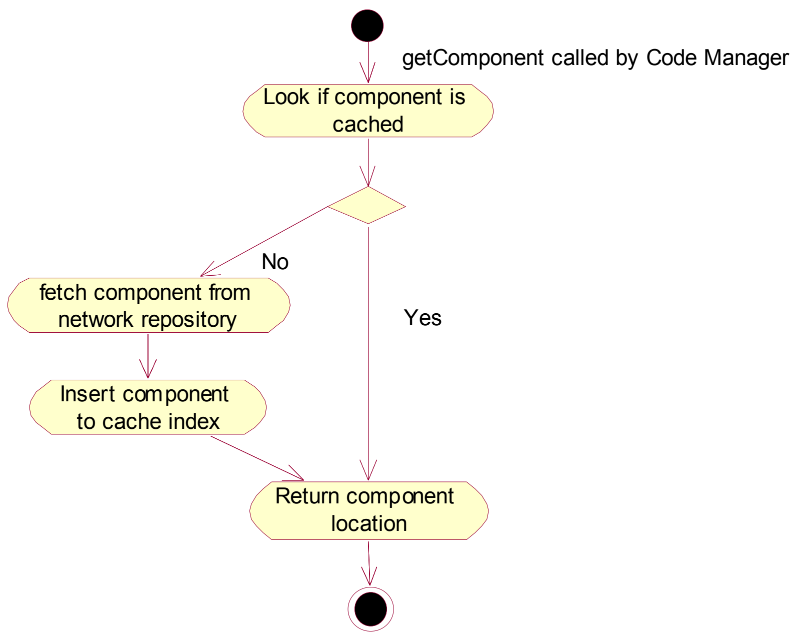

getComponent called by Code Manager
Look if component is cached
No
fetch component from network repository
Yes
Insert component to cache index
Return component location

Figure 76. Local Service Repository Activity diagram

The activity diagram describes the main operations performed when a component is requested by the code manager. First the local repository looks up in an index to see if the component is already stored locally. If the component is not cached, it is downloaded from the network-wide repository. After download, it is entered in the cache index and the local location of the component is returned to the code manager.

3.5.2.5 Service Creation Engine

The Service Creation Engine (SCE) [6] selects appropriate code modules to be installed on the node in order to perform the requested service functionality. The service creation engine matches service component requirements against node capabilities and performs the necessary dependency resolution. Since the service creation engine is implemented on each active node, active node manufacturers are enabled to optimise the mapping process for their particular node. In this way it is possible to exploit proprietary, advanced features of an active node. The selection of service components is based on service descriptors. Moreover, service descriptors describe how service components are bound to each other. This type of information is extracted by the SCE and passed to the code manager.

3.5.2.5.1 SCE use cases

Figure 77 shows the SCE use cases, which are described in this section.

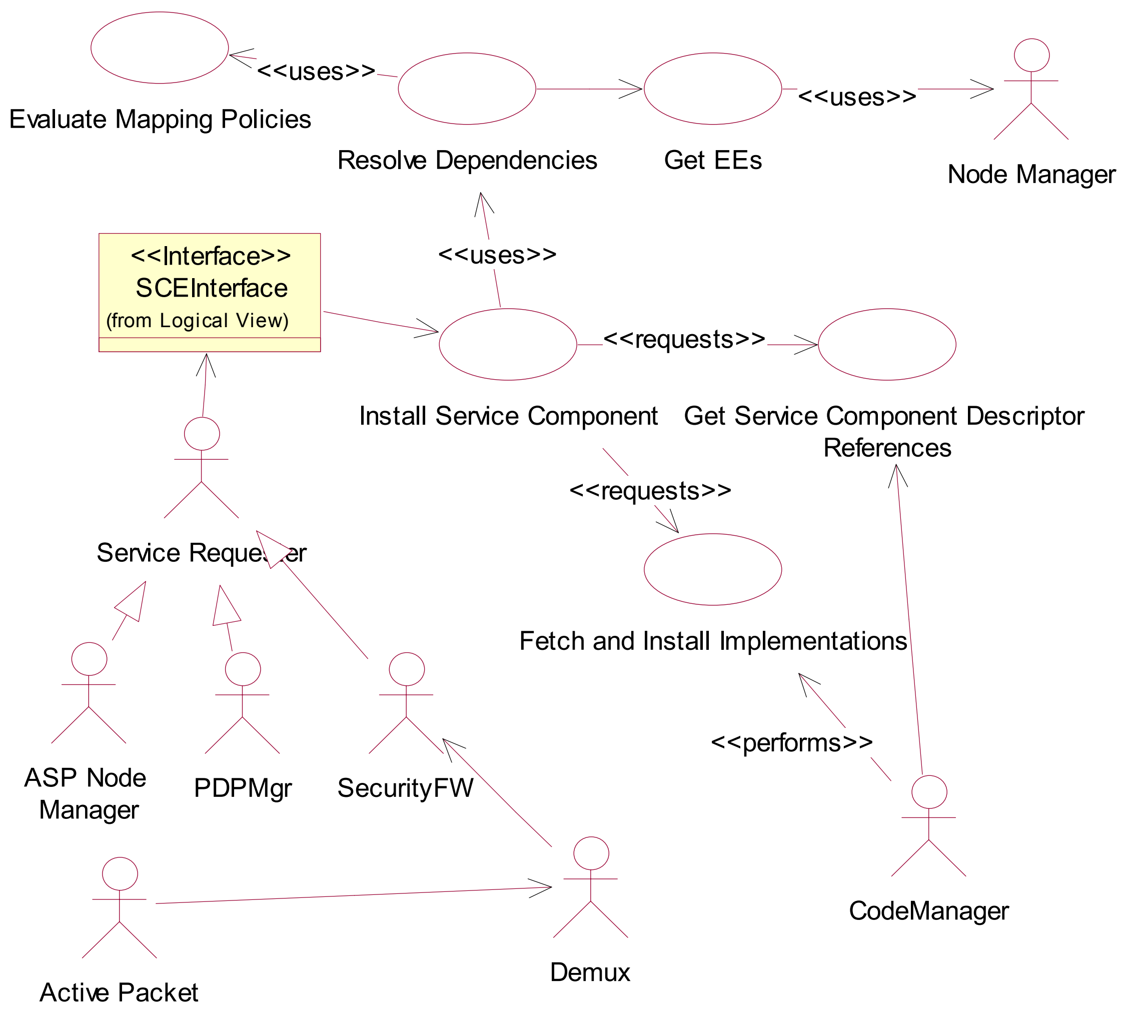


Figure 77. Service creation engine use cases

Install Service Component: The Install Service Component use case is offered to service requester. Different actors, such as ASP Node Manager, Demux (in case of an active packet), or the PDP Manager may take on this role. To carry out service component installation, interaction with the code manager is needed. Furthermore, this use case involves the Dependency Resolution use case.

Check Service: This use case is offered to the ASP Node Manager to check whether a service can be deployed on a specific node. To do so, the Dependency Resolution use case is involved.

Dependency Resolution: This use case is needed to determine the code modules needed to offer functionality defined by the service descriptors on a specific node. It is involved in both, the Install Service Component, as well as in the Check Service use cases described above. To perform this use case, interaction with Virtual Environment Manager and Code Manager is needed.

3.5.2.5.2 Interfaces and Implemented Operations

Listening: IDL for Service Creation Engine

```
interface SCE {

/**
* identifies required service components and selects compound
* implementations and implementations to be installed, based
* on the capabilities of a particular node.
* @returns a service runtime reference to the component manager
 * @param service_name well-known identifier of the behavioural characteristics
 * of the service to be installed.
 * @exception InstallationFailed is thrown if the installation did not succeed
 */

 ServiceRuntimeRef installServiceComponent(in ServiceComponentName service_name, VeID
   target_ve)
 raises (InstallationFailed);

 };
```

The SCE implements two public methods:

· `installServiceComponent`: This method performs node specific dependency resolution and identifies required code modules that implement the functionality specified by the service descriptor(s). Furthermore, from the service descriptors information is extracted describing the way code modules must be bound to each other. The download and installation of those modules is triggered using a method of the code manager. A reference to the service component is returned, which allows performing further management operations on it.

· `checkService`: This method checks whether the functionality specified by the service descriptor(s) can be implemented on a specific node, i.e. dependencies are resolved and it is verified that all necessary code modules exist. The code modules, however, are neither downloaded nor installed. A list of involved EE identifiers is returned.

3.5.2.5.3 Methodology

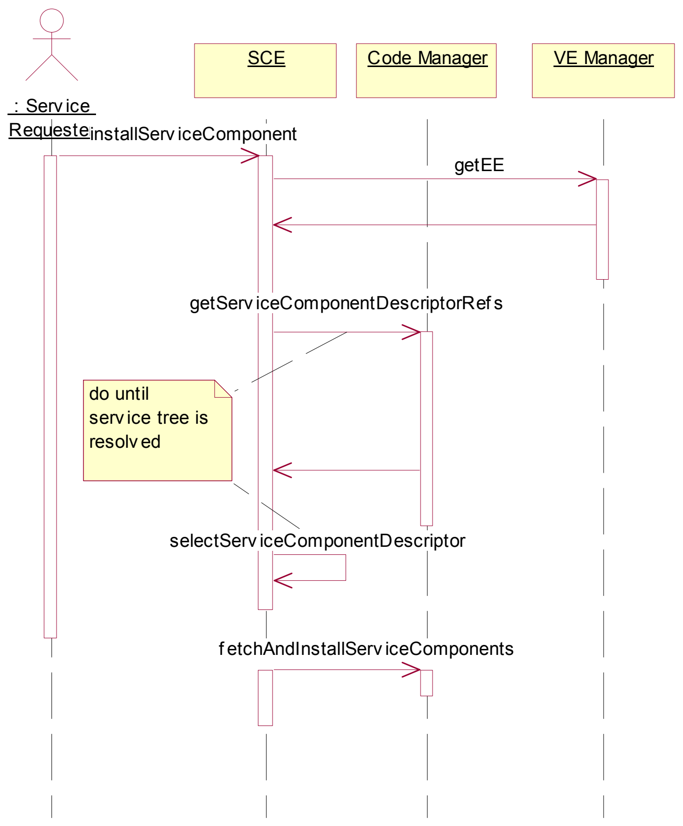

: Service Requester
SCE
Code Manager
VE Manager
installServiceComponent
getEE
getServiceComponentDescriptorRefs
do until service tree is resolved
selectServiceComponentDescriptor
fetchAndInstallServiceComponents

Figure 78. Service creation engine sequence diagram for "installServiceComponent" method

Install Service Component Flow: The SCE is responsible to map a service component name to code modules suitable to the local node environment.

The SCE starts with a service component name that stands for a specific type or functionality. Based on the service component name, the SCE requests a list of matching service component descriptors. From this list, the SCE selects – based on the available EEs - the appropriate service component descriptor. If a service component descriptor contains a non-empty list of service component names that it depends on, the resolution process continues in a recursive manner. A service component descriptor might contain a reference to a code module. If such a service component is selected by the SCE, the necessary information is stored in the *serviceComponents* data structure (see section 3.5.2.2).

The resolution process terminates when all dependencies are resolved. The binding between the code modules is determined based on the information available from the different service descriptors and stored in the *connectionInfos* data structure (see section 3.5.2.2). The SCE subsequently requests the download and installation of the compound implementations and implementations from the code manager. The necessary information is in the installation map, which is passed to the code manager, using the *fetchAndInstallServiceComponents* method.

Check Service Flow: The *checkService* flow is basically the same way as the *installServiceComponent* method, except that the code manager to trigger the download and installation of code modules is **not** contacted.

3.5.2.5.4 Extensions

The extension of the ASP with the Network ASP functionality led to internal changes in the Node ASP as well. These changes affect internal interfaces and the Service Creation Engine (SCE) functionality.

These modifications were necessary due to the partitioning of service information knowledge between the Network ASP and the Node ASP. Network ASP needs to pass requests to the Node ASP when element level service information is involved. As previously mentioned, this element level service information is only processed within the Node ASP so it has to undertake partial tasks where it has exclusive knowledge in.

The SCE was extended so that it can check if implementations exist for specific target environments found. Therefore the SCE can be used to check if a node is suitable for the deployment of a specific service, before actually initiating the service deployment process which implies service distribution, installation, instantiation and configuration. These initial checks into the suitability of the target environments reduce the potential waste of time and other resources that would result from complete failure of a service deployment.

A detailed application of these changes is described in section 3.5.1.1.1. They are reflected in the use case diagram of the SCE. The modified internal interfaces can be seen in the section 3.3.

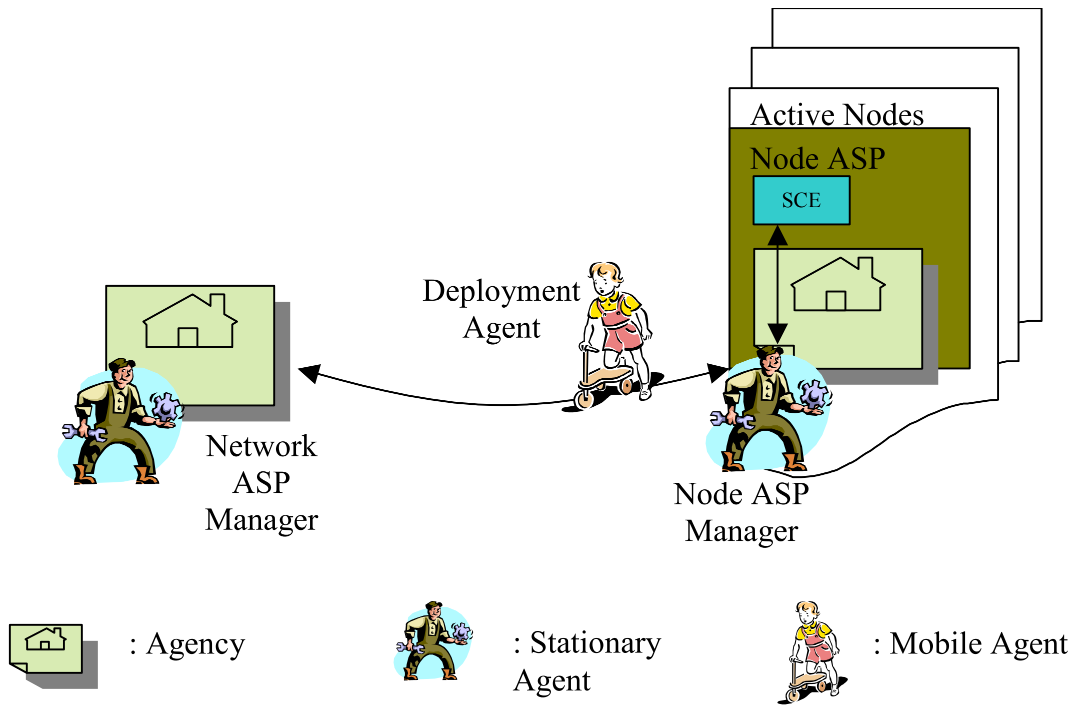


Figure 79. Validation of potential target environments

The Network ASP is involved in certain stages of the complete process of service deployment, performing operations that need service knowledge. The complete service deployment process also includes preparation stages like service release and requirements matching.

The Network ASP Manager provides service topology requirements that are defined as static service information within a network level service descriptor.

4. CONCLUSIONS AND FUTURE WORK

Since the last technical review, many improvements have been achieved regarding the FAIN **policy-based network management system**. A set of core components have been designed and implemented to build the hierarchically distributed architecture, through inheritance and enhancement to each management level. These core components include all common functionality to each level, i.e., all functionality related to the policy processing logic. Furthermore, we have added an extension mechanism that allows the management system to cope with new requirements unforeseen at design time.

At the element level, PDPs have being extended and enhanced to cope with the expected functionality. The PEPs have being designed and implemented as active service, running within FAIN active nodes. At the network level, the PBNM has been substantially extended to allow the creation/extension of a VAN by deploying a given service. The VAN can be under a unique administrative domain or encompass different domains.

VAN creation and extension processes make extensive use of the FAIN service management functionality designed and implemented at the network-levels of both the management system and the ASP system. These two systems interact to find the most appropriate service implementation based on the user (e.g., NIP, ANSP, or SP) requirements and the available resources, and to finally deploy the service, on the chosen FAIN active nodes.

Based on the FAIN Enterprise Model that advocates the deployment of virtual networks on top of the same network infrastructure, we have extended the concept of management by delegation to allow multiple management architectures to be instantiated and to function independently of each other. This has been achieved by using FAIN active node and its open facilities and interface.

In summary, we have achieved the following features:

- *Delegation of management* functionality: We cover this functionality in several ways: firstly, with the SP's access to ANSP management functionality (within the node and the element manager). The SP may also use its own code to manage allocated resources in order to offer a service. In order to allow the SP to do that the ANSP restricts the node interface offered to that code.
- *Creation of an active virtual network* for an SP: An SP obtains several isolated computational and communication resources, which conform a virtual environment in a group of, interconnect active nodes across the active network, thus creating an active virtual network.
- *Dynamic downloading of service-specific management components* from the SP: This was developed in order to be able to specifically manage the service it offers. This component is installed within the management stations and might interact with some of its components, e.g., the monitoring system in order to be able to make service decisions (e.g., for customer bandwidth reallocation).
- *Dynamic extensibility* of the management stations functionality is available by downloading new PDPs when they are needed. The dynamic installation of an active service within the node that is composed by two components: one running at the control plane and the other one in the data plane. The former controls the behaviour of the latter.

The **active service provisioning** system enables on-demand deployment of heterogeneous distributed component-based active services in the FAIN network.

The main features of the systems are:

- *Two-layered architecture:* The rationale for choosing this architecture was a separation of concerns in the service deployment problem space. Whereas the network-level ASP deals with network issues including identifying nodes of the target environment for a given service with regard to the topological service requirements and network Quality of Service requirements, the node-level ASP is concerned with node specific requirements, including technology and other service dependencies.
- *Heterogeneous active service support:* The ASP enables deployment of active services independently of the implementation technology they are based on. As long as a service is structured in terms of components and described using universal service descriptors defined in XML, it can be deployed using the ASP in the same way.
- *Multi-EE node support:* The ASP allows for deployment of active services on top of multi execution environment nodes. A service may consist of components to be deployed in different execution environment on a node. The decision as to which EE should be chosen depends on the execution capability and the availability of the suitable component implementation.
- *Deployment support for service components in different planes:* ASP is designed for deploying service code independent of the purpose. In the same way, it is possible to deploy components to run in management, control and data plane.
- *Hybrid two-phase process for the selection of a target environment:* The selection of active nodes suitable for a deployment of active services is designed as a hybrid of a centralized pre-section of the candidate nodes to be used for service deployment and a decentralized checking that the actual node capabilities on the candidate nodes suffice the service needs.
- *Universal service meta-information description:* The service descriptors are expressed in XML, a commonly-used SGML-based language standardized by W3C. By applying this language, the descriptors are easy to write for the service providers, easy to process by the programs (e.g. to generate it automatically by developing a service or to parse it) and, last but not least, as easy to extend. The common availability of the parsers also makes the software processing XML-based service descriptors easy to port to other platforms.
- *Binding of service components*: The FAIN ASP also supports binding service components forming a service. A service descriptor enables describing the way the components should be connected with each other and the node level ASP can interpret this information and perform the necessary actions.

The network level ASP has been significantly extended from its rudimentary form allowing for deploying active services on pre-selected number of nodes by adding to it logic enabling automatic selection of the target nodes based on matching the service requirements regarding the locations of the service components against the capabilities of the network available to the service provider at the deployment time. On the node level, the ASP code has profited from more thorough testing when performing the more advanced test case scenarios required for deploying more complex services, composed of components deployed on different active nodes distributed in the network

The future work will focus on providing improved support for service reconfigurability. Deployment algorithms and more optimised target environment selection algorithm will also be investigated.

5. REFERENCES

[1] FAIN Deliverable D5 “Revised Specification of Case Study Systems”, May 2002 – http://www.ist-fain.org

[2] Steven J. Metsker, “Design Patterns Java Workbook”, Addison Wesley, March 2002

[3] FAIN Deliverable D1: Requirements Analysis and Overall Architecture. FAIN Consortium, May 2001, pp. 11-18, http://www.ist-fain.org

[4] M.Solarski, M.Bossardt, T.Becker “Component-based Deployment and Management of Services in Active Networks”, IWAN’02, Zürich, CH, Dec. 2002.

[5] Larman C.: Applying UML and Patterns, 2nd Ed., Prentice Hall, ISBN 0-13-092569-1, 2002.

[6] Matthias Bossardt, Lukas Ruf, Rolf Stadler, Bernhard Plattner: A Service Deployment Architecture for Heterogeneous Active Network Nodes. Kluwer Academic Publishers, 7th Conference on Intelligence in Networks (IFIP SmartNet 2002), Saariselkä, Finland, April 2002.

[7] FAIN Project WWW Server – http://www.ist-fain.org

[8] K. Chan, et. al., “COPS Usage of Policy Provisioning”, IETF RFC 3084, March 2001.

[9] J.E. van der Merwe, S. Rooney, I.M. Leslie and S.A. Crosby, “The Tempest - A Practical Framework for Network Programmability”, IEEE Network, Vol 12, Number 3, May/June 1998, pp.20-28. http://www.research.att.com/~kobus/ docs/tempest_small.ps

[10] G. Goldszmidt and Y. Yemini. “Distributed Management by Delegating Mobile Agents”. In The 15th International Conference on Distributed Computing Systems, Vancouver, British Columbia, June 1995. http://www.cs.columbia. edu/~german/papers/icdcs95.ps.Z

[11] “Specification of Revised Case Study Systems”, FAIN Project Deliverable 5, http://www.ist-fain.org/deliverables/del5/d5.pdf.

[12] “Node OS Interface Specification”, AN Node OS Working Group, Larry Peterson, ed., November 30, 2001. http://www.cs.princeton.edu/nsg/papers /nodeos-02.ps

[13] M. Sloman, E. Lupu “Policy Specification for Programmable Networks” in Proceedings of IWAN99, 1999

[14] N. Damianou, N. Dulay, E. Lupu, M Sloman, “The Ponder Specification Language” Workshop on Policies for Distributed Systems and Networks (Policy2001), HP Labs Bristol, 29-31 Jan 2001. http://www.doc.ic.ac.uk/~mss/Papers/Ponder-Policy01V5.pdf

[15] Open Source Project OpenORB http://openorb.sourceforge.net/

[16] Distributed Management Task Force, Inc., "DMTF Technologies: CIM Schema V. 2.6", February 2002

[17] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, “Policy Core Information Model -- Version 1 Specification”, IETF Policy Working Group, RFC3060, February 2001.

[18] FAIN Deliverable D3, “Initial Specification of Case Study Systems”, May 2001.